Damiano Zanardini

Certified
Abstract Non-Interference -
Object-Oriented Code Validation
for Information Flow Security

Ph.D. Thesis

21 Aprile 2006

Università degli Studi di Verona
Dipartimento di Informatica
Advisor:
prof. Roberto Giacobazzi

Series N°: **TD-07-06**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy
This work is organized in seven chapters. The introduction, Chapter 1, illustrates the context and highlights the importance of security properties in the present day practice of software developing and analysis.

Chapter 2 introduces the various programming languages which are used throughout the remaining chapters (excluding Haskell, which implements the dependency calculus in Appendix A). It also provides the necessary background in lattice and fixpoint theory.

The original definition of Abstract Non-Interference and its foundational theories (in particular, Abstract Interpretation and the basics of Information Flow analysis) are presented in Chapter 3. Since we are dealing with the framework of code certification, the promising Proof-Carrying code architecture is also illustrated. Proof-Carrying code is the goal of the framework which is sketched in Chapter 6. Chapter 3 also presents, in its last section, recent research which is relevant to our purpose. In particular, security analysis for Object-Oriented programming languages is illustrated. Our focus will also be on the current research on modeling abstract properties.

The main part of this work begins with Chapter 4. The problem of representing abstract information flow security is here widely discussed. A type-based approach is first presented, which is an initial step towards automatizing Abstract Non-Interference. Then, a realistic approach to ANI is discussed. We provide a comprehensive definition of ANI for a Java-like language. For this purpose, we take advantage of the language mechanisms for enforcing correctness.

The definition of the previous chapter is applied in Chapter 5, which develops ANI analysis on source code programs by means of boolean functions. The information flow analysis relies on the calculus of abstract dependencies on data, which are computed by an static analysis algorithm. Dependencies are also considered in their relation with abstract program slicing.

Source code analysis is applied, in Chapter 6, to bytecode, in the direction of PCC-style code certification. Our security analysis is put in relation with
a Proof-Carrying code architecture, based on the bytecode level instead of native code programs.

Conclusions and future research directions are outlined in Chapter 7.
Saying thank you to a lot of people would sound to me as overemphasizing the importance of my work. Then, I'll try to be moderate...

First, I must be grateful to Prof. Roberto Giacobazzi for accepting me as one of his students and helping me during these fruitful years. This work would not have been possible without the help of Prof. Giuseppe Favretto. Thanks to people at Centro Docimologico, who were valuable partners and are still good friends.

I feel particularly indebted to Prof. Andrea Masini: his constant support and a couple of friendly conversations, more or less at the α and the ω of my PhD, definitely helped me in my efforts. (Not only) Young people at Ca' Vignal II have been to me an example with their spirit and skill. I’d like to mention here Isabella, Mila and Rosalba.

Adding the list of who, in my personal life, has been significant to me would be long and, probably, not very interesting. I only remember my parents. The others... indeed, they are important, and they know it.

D. Z.
Contents

Summary ................................................................. i
Acknowledgments ......................................................... iii
Preface ................................................................. IX

1 Introduction ............................................................. 1
  1.1 Software security ................................................... 1
  1.2 Confidentiality ..................................................... 2
  1.3 Relaxing security requirements ................................... 3
  1.4 Code Certification ................................................ 4
  1.5 The goal of this work ............................................. 4

2 Preliminaries ........................................................... 7
  2.1 Example languages ................................................ 7
    2.1.1 Imperative simple language ................................... 7
    2.1.2 Functional simple language ................................... 8
  2.2 The Object-Oriented Java-like language ....................... 9
    2.2.1 Syntax and semantics ....................................... 9
    2.2.2 Primitive types ........................................... 10
    2.2.3 Classes as singletons .................................... 10
  2.3 Java bytecode .................................................. 11
    2.3.1 Syntax and semantics .................................... 12
  2.4 Lattice and fixpoint theory .................................... 14
    2.4.1 Relations ................................................ 14
    2.4.2 Orderings and lattices .................................. 14
    2.4.3 Functions .............................................. 15
    2.4.4 Fixpoints ............................................. 15
### 3 Theoretical foundations and state of the art

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Abstract Interpretation</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Program analysis frameworks</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2 Principles of Abstract Interpretation</td>
<td>18</td>
</tr>
<tr>
<td>3.1.3 Abstract domains</td>
<td>21</td>
</tr>
<tr>
<td>3.1.4 Property transformers and completeness</td>
<td>22</td>
</tr>
<tr>
<td>3.1.5 Fixpoint approximation</td>
<td>23</td>
</tr>
<tr>
<td>3.1.6 Program properties</td>
<td>24</td>
</tr>
<tr>
<td>3.1.7 Typical (numerical) domains</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Information Flow Analysis</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Abstract Non-Interference</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1 Weakening Non-Interference</td>
<td>28</td>
</tr>
<tr>
<td>3.3.2 Attackers as abstract interpretations</td>
<td>28</td>
</tr>
<tr>
<td>3.3.3 Narrow and Abstract Non-Interference</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Program Slicing</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1 An algorithm for program slicing</td>
<td>31</td>
</tr>
<tr>
<td>3.4.2 Slicing and information flow</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Proof-Carrying Code</td>
<td>33</td>
</tr>
<tr>
<td>3.5.1 The PCC architecture</td>
<td>33</td>
</tr>
<tr>
<td>3.5.2 A novel approach to PCC architecture design</td>
<td>34</td>
</tr>
<tr>
<td>3.5.3 Application examples</td>
<td>35</td>
</tr>
<tr>
<td>3.6 The state of the art: Object-Oriented Information Flow analysis</td>
<td>36</td>
</tr>
<tr>
<td>3.6.1 Analysis with pointer aliasing</td>
<td>36</td>
</tr>
<tr>
<td>3.6.2 Abstract dependencies</td>
<td>39</td>
</tr>
<tr>
<td>3.6.3 JVM-like programming languages</td>
<td>40</td>
</tr>
<tr>
<td>3.6.4 Code certification</td>
<td>41</td>
</tr>
<tr>
<td>3.6.5 Mathematical formalisms</td>
<td>42</td>
</tr>
<tr>
<td>3.6.6 Proving Abstract Non-Interference</td>
<td>43</td>
</tr>
</tbody>
</table>

### 4 Representing Abstract Non-Interference

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Security types</td>
<td>46</td>
</tr>
<tr>
<td>4.2.1 Representing abstract Information Flow with types</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2 Higher-Order security types and type inference</td>
<td>48</td>
</tr>
<tr>
<td>4.2.3 Stack-based security types and type inference</td>
<td>51</td>
</tr>
<tr>
<td>4.2.4 Brief discussion</td>
<td>51</td>
</tr>
<tr>
<td>4.3 A realistic approach to Information Flow</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1 Which definition of Non-Interference</td>
<td>52</td>
</tr>
<tr>
<td>4.3.2 What is an NI-secure program</td>
<td>53</td>
</tr>
<tr>
<td>4.4 Class-Oriented Abstract Non-Interference</td>
<td>57</td>
</tr>
<tr>
<td>4.4.1 Class hierarchies as abstract domains</td>
<td>58</td>
</tr>
<tr>
<td>4.4.2 Class intersection</td>
<td>59</td>
</tr>
<tr>
<td>4.4.3 Domains and program code</td>
<td>59</td>
</tr>
<tr>
<td>4.4.4 Class extension</td>
<td>60</td>
</tr>
<tr>
<td>4.4.5 Advantages and limitations</td>
<td>61</td>
</tr>
</tbody>
</table>
5 Program analysis for ANI ........................................... 65
  5.1 Preliminary definitions ........................................ 65
    5.1.1 Methods ................................................... 65
    5.1.2 Type environments ....................................... 66
    5.1.3 Program identifiers ..................................... 67
  5.2 Abstract dependency calculus ................................ 69
    5.2.1 Preliminary notions ..................................... 69
    5.2.2 Abstract Dependencies ................................. 71
    5.2.3 Computing an expression on its inputs ............... 72
    5.2.4 Increasing efficiency .................................. 74
    5.2.5 The algorithm ........................................... 76
    5.2.6 Discussion ............................................... 82
  5.3 Boolean functions .......................................... 83
    5.3.1 Models and Abstract Non-Interference ................ 84
  5.4 Source code analysis via boolean functions ............... 85
    5.4.1 Analysis of statements ................................ 85
    5.4.2 Whole-program analysis ............................... 87
    5.4.3 Widening ................................................ 89
  5.5 Correctness and further discussion ......................... 90
    5.5.1 The ordering on \( \mathfrak{A} \)-states ..................... 91
    5.5.2 Monotonicity ............................................. 91
  5.6 The link to Program Slicing ................................ 94
    5.6.1 Computing abstract slices ............................ 95
    5.6.2 Obtaining consistent slices ......................... 96
    5.6.3 Discussion ............................................... 97

6 Code certification for ANI ...................................... 99
  6.1 Analyzing bytecode programs ............................... 100
    6.1.1 Bytecode boolean functions .......................... 100
    6.1.2 A simple source code compiler ....................... 100
    6.1.3 The Control Flow Graph ................................ 101
    6.1.4 Analysis of blocks .................................... 104
    6.1.5 Expressions .............................................. 107
    6.1.6 Program analysis ....................................... 108
  6.2 The relation between source code and bytecode .......... 108
    6.2.1 Variable assignment .................................... 108
    6.2.2 Field assignment ....................................... 109
    6.2.3 Conditional statement ................................ 109
    6.2.4 Non-compiled bytecode ................................ 110
  6.3 Towards a Proof-Carrying code architecture .............. 110
    6.3.1 What the code user receives ......................... 110
    6.3.2 What the code user should verify ................... 110

7 Conclusions ..................................................... 113
VIII  Contents

A  The abstract dependency calculus: Haskell Implementation 115

References ......................................................... 119

Index ............................................................... 125

Sommario .............................................................. 133
Preface

This work is the result of my investigation on abstract security properties with respect to code validation. Among its goals, the main one was to provide the first steps towards automatizing the design and checking of safe programs. Outside of this preface, I will refer to myself as we, even if, up to now, nothing has been published as a joint work.

Part of this work, in particular Section 4.2, has already been published [73, 74]. Despite the advice of one of the referees, I chose to leave this section in its original place, although it may seem a bit incoherent with respect to the following discussion.

Future plans include publishing some other parts of the last chapters, in particular the class-oriented notion of Abstract Non-Interference, the calculus of abstract dependencies, and the algorithms for code certification via boolean functions.

For brevity, index entries are only included once (their first occurrence) for every section; however, this rule could have been violated somewhere.
1

Introduction

The mathematics is not there till we put it there.

Sir Arthur Eddington (1882 - 1944), *The Philosophy of Physical Science*

I have suffered a great deal from writers who have quoted this or that sentence of mine either out of its context or in juxtaposition to some incongruous matter which quite distorted my meaning, or destroyed it altogether.

Alfred North Whitehead (1861 - 1947)

This chapter describes the context of this work. We discuss the importance of confidentiality in present-day software and outline how relaxing security requirements might lead to mere practical frameworks. Moreover, the goal of this work is introduced, together with its relation with the program certification.

1.1 Software security

The importance of software systems is becoming larger and larger in everyday life. An enormous quantity of data is manipulated by information systems, whose complexity requires sophisticated tools for design and checking. Due to the importance of manipulated data, programs must fulfill several security requirements in order to be safely executed. The execution of software must not negatively alter the surrounding environment (for example, by provoking system crashes or modifying/deleting relevant information).

Today, programs are much too large to be checked manually, in a reasonable amount of time and resources, by programmers. Consequently, code designers need automatic, flexible and efficient tools for checking security properties of programs. Checking the safety of software is one of the main goals of
1 Introduction

Static Analysis, a class of techniques for analyzing the behavior of programs without executing them. Static Analysis directly inspects the program code for inferring approximated properties.

A security policy is a formalization of security requirements. Static Analysis is a useful tool for checking the adherence of programs to some security policy; it can also drive the code design process, by automatically assisting the process of software production under the light provided by the policy.

1.2 Confidentiality

An important class of security properties is related to the protection of confidential data. It is often the case that a part of the data which are manipulated by an information system must be protected by malicious users interacting with the system. An attacker can be interested in private data. If they are passive, attackers cannot modify the attacked system; yet, they can observe the execution of programs in order to acquire information about data they manipulate.

Making the access to data secure

Observing the execution of programs amounts to watching their behavior on a given input. Attackers can only see a portion of program data, the public part. Their observations are limited to the public part of input and output data; it can possibly include auxiliary information, such as the execution time or the amount of allocated memory.

Access Control techniques regulate the access to protected data. Their goal is to prevent users from reading forbidden data. A piece of data is forbidden for a user if accessing to it requires more privileges than those which are possessed by the user. It is quite a common situation, in today software systems, to have some users (administrators and/or developers) which are allowed to execute programs or manipulate data, forbidden for normal users. For example, some web sites have several privilege levels: visitors, free members, paying members and administrators. The security policy should make sure that a user cannot access data which are reserved to users with more privileges than his/hers.

Regulating the propagation of secret data

Access control prevents the direct access to private information. On the other hand, users can freely access non-confidential data. Due to some (usually undesired) aspects of the program behavior, it can be the case that untrusted (possibly malicious) users are able to acquire private information without directly accessing to protected data. This can happen if
attackers have some knowledge of the operations performed by the program, that is, they are able to correctly interpret their observation of the program semantics; and

- the final value of non-protected data depends on protected information, so that it is possible to guess the value of private data by only reading non-confidential information.

The first requirement may seem, at a first glance, quite unrealistic, since the semantics of many commercial programs is protected by copyright and not freely observable. However, decompiling techniques allow to guess information about the program semantics by re-building the program source code. More importantly, a free knowledge of program behavior is made perfectly realistic by the growing importance of open source software: many systems rely on software whose behavior is public, that is, can be freely observed by anyone.

A dependency between program data generates a information flow. Information is said to flow from $c_1$ to $c_2$ if $c_2$ depends on $c_1$, that is, if the initial value of $c_1$ is relevant to the final value of $c_2$. In this case, it is possible to guess something about $c_1$ by merely observing $c_2$. Information Flow analysis checks whether information flows dangerously from private to public data. This amounts to say that, in order to preserve confidentiality, a program should not generate forbidden flows, that is, the possibility to acquire protected information by watching public data. This security property is called Non-Interference; the private and the public part of program data are said not to interfere if there is no transfer of information across the private-public boundary.

1.3 Relaxing security requirements

If a program satisfies Non-Interference, then its publicly observable behavior is completely independent with respect to protected data. From the semantic point of view, private information could also be completely different, or even disappear, without resulting in any change in the observable behavior. Actually, this complete separation between private and public data may seem too strong a requirement. In facts, we could ask whether such a program is practically equivalent (regarding untrusted users) to another one which does not use private data at all.

Allowing selected flows

This question led computer scientists to several attempts towards the weakening of the standard Non-Interference notion. Weakening Non-Interference amounts to allow some selected flows from private to public data. Flows are permitted in order to model realistic frameworks where interference must be regulated but not totally forbidden.
Aggregate data (that is, those which are obtained by some computation on arrays of data, such as average, sum or standard deviation) on private information are a typical example of interference which is often considered as harmless; yet, the public result clearly depends on protected data. Some approaches allow everyone to read aggregate data (for example, the average of monthly salaries in an enterprise), provided that no knowledge about particular data can be inferred (for example, the salary of the employee Xx Yy).

Limiting the power of attackers

Abstract Non-Interference approaches this problem with a different perspective: attackers are assumed to have a reduced observational power, so that they can only observe program execution up to some degree of approximation (formalized by means of the theory of Abstract Interpretation). Dangerous ‘nows are permitted as long as they cannot be observed and exploited by attackers.

1.4 Code Certification

Developing secure programs is one of the main goals of program analysis techniques, such as Static Analysis. A programmer is usually interested in producing code which fulfills some given security requirements. Yet, it can happen that the code designer writes (by error or malice) programs which, when executed in an external system, disclose confidential data which are contained in the system. In this case, a user, which wants to execute software received from an untrusted source, can possibly (and involuntarily) release some of his or her secrets by running this malicious code.

The growing importance of Web-based communications makes the problem of executing safe code crucial: most users do not pay enough attention to what they download from the World Wide Web and run on their systems. Consequently, mechanisms for automatically deciding when a program can be safely executed represent an utterly important tool for improving the security of systems. Proof-Carrying code follows this direction: the guarantees for the safety of a program are contained in the program itself, so that the user can check the security properties (that is, certificate the program) before executing the code. In principle, Proof-Carrying code architecture can deal with a wide variety of security policies; among them, confidentiality.

1.5 The goal of this work

When dealing with abstract information flow properties such as Abstract Non-Interference, we must face the problem of representing abstract properties of data and their related operations.
1.5 The goal of this work

Our purpose is to discuss the problem and provide a definition of Abstract Non-Interference which is more practical and easier to be used in a realistic framework. We consider an Object-Oriented programming language, modeled on Java, and describe data properties as classes. In this approach, the analysis of properties can be reduced, up to a certain extent, to class analysis.

We try to suggest an algorithm for the computation of abstract data dependencies. The calculus of abstract dependencies is a semantic task, then it is, in the general case, undecidable. The provided algorithm computes an over-approximation of dependencies, relying on standard static analysis type-based techniques. Abstract dependencies are crucial in defining approximated information flow properties; they are also relevant to the abstract version of program slicing.

Some ideas for a certification architecture are included; in particular, the algorithm for checking the security properties of source code is transposed to the bytecode low-level language. However, in order to obtain a realistic Proof-Carrying code framework, some further work is needed.
Preliminaries

There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things.

Niccolò Machiavelli (1469 - 1527), The Prince

Quotation, n: The act of repeating erroneously the words of another.

Ambrose Bierce (1842 - 1914), The Devil’s Dictionary

This chapter introduces notions which are useful in the rest of the work. The IMP, FUN and JL programming languages are illustrated. IMP and FUN are used in simple examples; JL is the target language of Information Flow analysis. Also, a description of the Java bytecode subset obtained by compiling JL programs is provided. Finally, we introduce the theoretical background which is necessary to understand Abstract Interpretation and related topics.

2.1 Example languages

In this work, some examples will be provided, which use simple programming languages. An imperative and a functional language will be used. In both cases, \( P(x) \) is a shorthand for \( \llbracket P \rrbracket(x) \) and stands for the result (output) of executing or evaluating the program \( P \) on the input \( x \).

2.1.1 Imperative simple language

This language can be often found in the literature as the IMP language. Integers are the only data type; input and output data are stored in variables \( x, y \in \mathbb{V} \), whose life covers the entire program execution. The set of variables is
unlimited; if not differently specified, it is intended that the input and output
of a program are, respectively, the initial and final values of all variables.
Basic statements are skip and variable assignment. Statements $s \in S$ can be
sequentially concatenated, chosen by means of conditional and iterated in a
loop; no procedures are provided. Expressions $E$ are simple arithmetical terms
obtained by combining numbers $\mathbb{Z}$ and operators.

\[
E ::= Z \mid V \mid E + E \mid E * E \mid (E) \mid \ldots
\]
\[
B ::= \text{true} \mid \text{false} \mid E == E \mid E <= E \mid B \text{ and } B \mid \ldots
\]
\[
S ::= \text{skip} \mid V := E \mid S ; S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{while } B \text{ do } S
\]

A program state is the memory configuration, that is, the values of variables
at a given time. The semantics $\llbracket \cdot \rrbracket$ is straightforward and will be omitted.

### 2.1.2 Functional simple language

The functional language FUN is a kind of typed lambda-calculus with $\text{int}$ as
the only basic type and the arrow type constructor $\rightarrow$.

#### Syntax

Functional expressions can be obtained by means of lambda-abstraction, ap-
lication, conditional and recursion. Arithmetical operations are shorthands
for their corresponding lambda terms.

\[
C ::= 0, 1, 2, 3 \ldots \quad \text{constants}
\]
\[
V ::= x, y, F \ldots \quad \text{variables}
\]
\[
E ::= C \mid V \mid E + E \mid \lambda V . E \mid E(E) \mid \mu V . E \mid \text{if } E \text{ then } E \text{ else } E
\]

#### Semantics

Semantic domains have a bottom element $\bot$; for functions, the bottom element
is $\lambda V . \bot$. The denotational semantics $\llbracket \cdot \rrbracket : E \mapsto S$ follows the call-by-value
evaluation rule.

\[
n \in \mathbb{Z}_\bot \triangleq \mathbb{Z} \cup \{\bot\} \quad \text{integers}
\]
\[
f \in \mathcal{U} \triangleq \mathbb{Z}_\bot \cup [\mathcal{U} \mapsto \mathcal{U}] \quad \text{values}
\]
\[
W \triangleq \{\omega\} \quad \text{wrong value}
\]
\[
\varepsilon \in \mathcal{E} \triangleq \mathcal{V} \mapsto \mathcal{U} \quad \text{environments}
\]
\[
\phi \in \mathcal{S} \triangleq \mathcal{E} \mapsto \mathcal{U} \quad \text{semantic domains}
\]
\[ [n]_e \overset{\Delta}{=} n \]
\[ [x]_e \overset{\Delta}{=} e(x) \]
\[ [x + y]_e \overset{\Delta}{=} [x]_e + [y]_e \]
\[ [\lambda x. e]_e \overset{\Delta}{=} \lambda v. [e]_{e[x \leftarrow v]} \]
\[ [\mu F. e]_e \overset{\Delta}{=} \text{LFP} \left( \lambda v. [e]_{e[F \leftarrow v]} \right) \]
\[ [e' = e'']_e \overset{\Delta}{=} ([e']_e = [e'']_e) \]
\[ [[\text{if } e \text{ then } e' \text{ else } e'']_e \overset{\Delta}{=} (\text{if } e_1 = e_2) \rightarrow (\text{if } e' \leftarrow e''_e) \]

Type checking is implicit: if a type error arises, expressions evaluate to \( \omega \).

The type system is the monomorphic type system of simply-typed lambda-calculus.

2.2 The Object-Oriented Java-like language

The Object-Oriented language JL is a subset of the Java [41] programming language, without advanced features such as concurrency and exceptions.

Remark 2.1. With abuse of notation, here and in the following the terms class and type will be used interchangeably when dealing with Java or JL.

2.2.1 Syntax and semantics

The language syntax is modeled on Java. Expressions and statements are defined as:

\[ \begin{align*}
v &\in V \\
f &\in F \\
x &\in I = V \cup \{v.f \mid v \in V, f \in F\} \\
m &\in M \\
C, D &\in T \\
E ::= I \\
| new T(E, \ldots, E) \\
| V.M(E, \ldots, E) \\
S ::= \text{skip} \\
| I = E \\
| V.M(E, \ldots, E) \\
| S; S \\
| \text{if } (B) S \text{ else } S \\
| \text{while } (B) S \\
| \text{return } E
\end{align*} \]

Identifiers cannot take the form \( x.f.g \). Actually, sequences of field references (dots) are not necessary; for example, the assignment

\[ x.f.g = e \]
can be replaced by adding a new local variable \( t \):

\[
\begin{align*}
  t &= x.f; & t.g &= e
\end{align*}
\]

This is correct since, after the first statement, \( t \) happens to be the object in the \( f \) field of \( x \). On the other hand, the sequence

\[
\begin{align*}
  t_1 &= x.f; & t_2 &= t_1.g; & t_2 &= e
\end{align*}
\]

seems to be an good alternative, with the benefit of ruling out dotted identifiers on the left side of assignments. However, we can be easily see that it does not work since the second assignment to \( t_2 \) invalidates the effects of the first one. Similarly, method invocation can be limited to \( v.m(...) \) for some variable \( v \). Boolean expressions \( b \in \mathbb{B} \) are simply objects with Boolean type. The language semantics is, basically, the Java semantics for the corresponding language subset.

### 2.2.2 Primitive types

In JL, Java primitive types int and boolean are replaced by primitive classes Integer and Boolean. Actually, replacing primitive types with primitive classes does not preserve the language semantics. In practice, operations as the creation of new objects involve using primitive values, as in new Integer(5). We choose to postulate the existence of an infinite set of Integer and Boolean constants, such as Boolean true, Integer two etc. Logical and arithmetical operations are also provided in form of methods. This hypothesis on language design allows to manipulate all program data as objects.

### 2.2.3 Classes as singletons

The Information Flow analysis described in Chapters 4 and 5 is type-based: the type of program data is the only relevant thing when computing abstract properties. In facts, it is its abstract property. The analysis is parametric on the power of attackers, that is, on the precision of properties.

Standard Non-Interference (that is, when attackers can see the actual value of data, see Section 3.2) is a special case of this general framework. Since, in the non-abstract case, every value can be distinguished, we need to transpose this condition to types. This amounts to making every value distinguishable by its type, that is, defining classes containing one single object (singleton classes).

**Example.** The most precise (concrete) properties on numerical values have the form to be 5, to be 6 etc. Similarly, the properties to be true and to be false are the most precise properties on booleans. We want to declare classes representing those properties, that is, containing one single object. They are declared as subclasses of primitive classes.
As we see, methods are also specialized in new classes. In practice, singleton classes represent constants. There is no need to have internal members such as the numerical value, since this information is already contained in types. Clearly, an infinite set of classes should be declared, together with suitable method definitions.

Such a reformulation of program data, in which classes are ubiquitous, is unrealistic, due to the huge number of class and method declarations. However, it has a theoretical meaning: it puts standard Information Flow analysis into the properties as classes framework. Indeed, singleton classes allow to express the operations on concrete values in a type-based way.

2.3 Java bytecode

Compiling a Java program results in a bytecode program. This is the form in which programs are executed on different platforms by means of the Java Virtual Machine [52].
The Java Virtual Machine is the cornerstone of the Java platform. It is the component of the technology which is responsible for its hardware- and operating system- independence, the small size of its compiled code, and its ability to protect users from malicious programs. The JVM has an instruction set and manipulates the memory at runtime. It knows nothing of the Java programming language, only of a particular binary format, the .class file format. Java does not even need to be the source language; in facts, implementors of other high-level languages are turning to the Java Virtual Machine as a delivery vehicle for their languages. A .class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

2.3.1 Syntax and semantics

A .class file contains information about a class and its members. The code of a method lies in the Code attribute of some method_info descriptor in the .class file. It consists of a sequence of bytecode instructions, that is, byte strings whose first byte is the opcode and the rest identifies parameters.

States and frames

Instructions are executed on states which contain local information, global constants and an operand stack (which is empty at the beginning of the method) to perform operations. A method is associated to a frame:

\[
X \in X_{\text{loc}} : I \mapsto V \quad \text{local variables}
\]

\[
O \in O : \text{stack of } V \quad \text{operand stacks}
\]

\[
R \in R : I \mapsto V \quad \text{runtime constant pools}
\]

\[
F = (X, O, R) \in F : X_{\text{loc}} \times O \times R \quad \text{frames}
\]

Frames are put in a framestack \(FS \in S_F\); in the following, when no pushing/popping is done on the framestack, only the frame on top of the stack is shown.

Instructions

The semantics for most bytecode instructions is given. In \(\langle (X, O, R) \rangle\), the frame \((X, O, R)\) is on top of the framestack and the other frames are not visualized. \(v : O\) is the operand stack obtained by pushing \(v\) onto \(O\). \(X[j]\) is the local variable with index \(j\).
\[\text{aconst\_null}\] \(\langle (X, O, R) \rangle = \langle (X, \text{null} : O, R) \rangle\)

\[\text{dup}\] \(\langle (X, v : O, R) \rangle = \langle (X, v : O, R) \rangle\)

\[\text{nop}\] \(\langle (X, O, R) \rangle = \langle (X, O, R) \rangle\)

\[\text{pop}\] \(\langle (X, v : O, R) \rangle = \langle (X, v : O, R) \rangle\)

\[\text{swap}\] \(\langle (X, v_1, v_2 : O, R) \rangle = \langle (X, v_2 : v_1 : O, R) \rangle\)

\[\text{aload}\] \(j\) \(\langle (X, O, R) \rangle = \langle (X, [j] : O, R) \rangle\)

\[\text{astore}\] \(j\) \(\langle (X, v, O, R) \rangle = \langle (X, [j \leftarrow v], O, R) \rangle\)

\[\text{getfield}\] \(j\) \(\langle (X, v : O, R) \rangle = \langle (X, u : O, R) \rangle\)

\[\text{getstatic}\] \(j\) \(\langle (X, O, R) \rangle = \langle (X, u : O, R) \rangle\)

\[\text{putfield}\] \(j\) \(\langle (X, v : O, R) \rangle = \langle (X, O, R) \rangle\)

\[\text{putstatic}\] \(j\) \(\langle (X, v, O, R) \rangle = \langle (X, O, R) \rangle\)

In the rules for putfield and putstatic, modifications to instance or static data on the heap are not shown. In the rule for getfield \(j\) (respectively, getstatic \(j\)), after the symbolic field reference \(R[j]\) has been resolved, \(u\) is the value of the instance (resp. static) field \(R[j]\) of the object referenced by \(o\) (resp. in the resolved class).

\[\text{invokestatic}\] \(j\) \(\langle (X, v_1, \ldots, v_k, O, R) \rangle = \langle (X'[k \leftarrow v_k], \emptyset, R') \rangle \over \langle X, O, R \rangle\)

\[\text{invokevirtual}\] \(j\) \(\langle (X, v_1, \ldots, v_k, O, R) \rangle = \langle (X'[k \leftarrow v_k], \emptyset, R') \rangle \over \langle X, O, R \rangle\)

The pushed frame is (i) for invokevirtual, the one corresponding to the class method; (ii) for invokevirtual, the instance method of \(o\). After the invoked method has terminated its execution, the final state is \(\langle X, u : O, R \rangle\) if the value \(u\) is returned, \(\langle X, O, R \rangle\) otherwise.

\[\text{areturn}\] \(\langle (X, O, R) \rangle = \langle (X', v : O', R') \rangle\)

\[\text{return}\] \(\langle (X, O, R) \rangle = \langle (X', O', R') \rangle\)

\[\text{ldc}\] \(j\) \(\langle (X, O, R) \rangle = \langle (X, R[j] : O, R) \rangle\)

\[\text{new}\] \(j\) \(\langle (X, O, R) \rangle = \langle (X, o : O, R) \rangle\)
2.4 Lattice and fixpoint theory

This section introduces basic notions on relations, functions, orderings and
fixpoints. Its main use is in defining the Abstract Interpretation theory in
Section 3.1.

2.4.1 Relations

Let \( S, T \) and \( U \) be sets. The powerset \( \mathcal{P}(S) \) is the set of all subsets of \( S \):
\[
\mathcal{P}(S) = \{X \mid X \subseteq S\}.
\]
The cartesian product \( S \times T \) is the set of pairs \( (s,t) \) such that \( s \in S \) and \( t \in T \). A binary relation on \( S \times T \) is a subset of the cartesian product: \( R \in \mathcal{P}(S \times T) \). Similarly, a k-ary relation is a subset of \( S_1 \times \cdots \times S_k \). A binary relation \( R \) on \( S \times S \) can be:
- reflexive if \( \forall s \in S. \ (s,s) \in R \)
- transitive if \( \forall s,t,u \in S. \ ((s,t) \in R \land (t,u) \in R) \Rightarrow (s,u) \in R \)
- symmetric if \( \forall s,t \in S. \ (s,t) \in R \Leftrightarrow (t,s) \in R \)
- antisymmetric if \( \forall s,t \in S. \ ((s,t) \in R \land (t,s) \in R) \Rightarrow s = t \).

2.4.2 Orderings and lattices

A partial order \( \sqsubseteq \) (or simply \( \sqsubseteq \) when no confusion arises) is a binary relation on \( S \) which is reflexive, transitive and antisymmetric. Usually, \( s \sqsubseteq t \) is a shorthand for \( (s,t) \in \sqsubseteq \). We write \( s \sqsubset t \) if \( s \sqsubseteq t \) and \( s \) and \( t \) are distinct.

Partial orders are also represented by the symbols \( < \) or \( \leq \).

Let \( S(\sqsubseteq) \) be a poset (shorthand for partially ordered set), that is, a set \( S \) with a partial order \( \sqsubseteq \) on its elements. The upper bound of a subset \( X \) of \( S \) is an element \( u \in S \) which is greater or equal than all elements of \( X \) (\( \forall t \in X. \ t \sqsubseteq u \)). We write \( s \sqsubset t \) if \( s \sqsubseteq t \) and \( s \) and \( t \) are distinct.

Partial orders are also represented by the symbols \( \leq \) or \( \geq \).

An ascending chain (or, equivalently, an increasing chain) is a subset of \( S \) such that \( \sqsubseteq \) is a linear order on \( X \), that is, any two elements of \( X \) are comparable: \( \forall x',x'' \in X. \ x' \sqsubseteq x'' \lor x'' \sqsubseteq x' \). A complete partial order (cpo) is a poset such that every ascending chain has a least upper bound. Given a poset \( S \), its Moore closure \( \mathcal{M}(S) \) is the set \( \{ \sqcap X \mid X \subseteq S\} \). \( S \) is a Moore family if \( \mathcal{M}(S) \) is a Moore family for both glb and lub).

A complete lattice \( L(\sqsubseteq, \sqcap, \sqcup, \bot, \top) \) is a poset such that \( \sqcap X \) and \( \sqcup X \) exist for every \( X \subseteq L \) (then, it is also a Moore family for both glb and lub). \( \sqcap \) and \( \sqcup \) are the top (maximum) and bottom (minimum) elements; they are defined as, respectively, \( \sqcup L \) and \( \sqcap L \).
2.4 Lattice and fixpoint theory

2.4.3 Functions

A function $\varphi : S \rightarrow T$ is a particular binary relation on $S \times T$, having the property that, for every $s \in S$, there exists at most one $t \in T$ such that $(s, t) \in \varphi$. Such an element $t$ can be written, for example, as $\varphi_s$, $\varphi(s)$ or $\varphi[s]$. If $\varphi : S \rightarrow T$, it is said that $\varphi$ maps $S$ to $T$ or, equivalently, that $\varphi$ is a map of $S$ to $T$. Functions can have more than one parameter, that is, be a subset of non-binary relations on $S_1 \times \cdots \times S_k \times T$. If $\varphi : S_1, \ldots, S_k \rightarrow T$, then $k$ is said to be its arity, that is, the number of its parameters. For $\varphi : S \rightarrow T$ and $X \subseteq S$, the expression $\varphi(X)$ often stands for $\{\varphi(x) \mid x \in X\}$. This set is said to be the image of $X$ by $\varphi$.

A function $\varphi : S \rightarrow T$ is total if its value is defined for every value $s$ of the parameter (or, for every $s_1, \ldots, s_k$ if $\varphi$ has arity $k$): $\forall s \in S. \exists t \in T. (s, t) \in \varphi$; otherwise, $\varphi$ is said to be partial. It is monotone if $s_1 \leq s_2$ implies $\varphi(s_1) \leq \varphi(s_2)$. $S_{\rightarrow} T$ is the set of all monotone functions from $S$ to $T$.

A function is upper-(lower-)continuous if it preserves existing least upper (greatest lower) bounds of ascending (descending) chains: $\varphi(\bigvee_X \leq \bigvee_T \varphi(X))$ (resp. $\varphi(\bigwedge_X \geq \bigwedge_T \varphi(X))$), where $\varphi : S \rightarrow T$ and $X \subseteq S$ is an ascending (descending) chain. Continuity implies monotonicity.

A function $\varphi : S \rightarrow S$ is extensive if its result is greater than its parameter: $\forall s \in S. s \subseteq \varphi(s)$. Its dual notion is reductivity. Also, $\varphi$ is idempotent if $\forall s \in S. \varphi(\varphi(s)) = \varphi(s)$.

If $s' \neq s''$ implies $\varphi(s') \neq \varphi(s'')$, then $\varphi : S \rightarrow T$ is said to be injective, or an injection of $S$ in $T$. Moreover, if $\forall t \in T. \exists s. \varphi(s) = t$ holds, then $\varphi$ is surjective, or a surjection of $S$ on $T$. A bijection is both injective and surjective.

Given two posets $S$ and $T$, the map $\varphi : S \rightarrow T$ is an order isomorphism (or simply an isomorphism, when no confusion arises with isomorphisms on binary operations) if it is a bijection and preserves the order of $S$ in $T$, that is, $\varphi(s_1) \leq \varphi(s_2)$ if and only if $s_1 \leq s_2$. $S$ and $T$ are said to be isomorphic.

Given two functions $\varphi'$ and $\varphi''$ mapping $S$ to $T$, when no confusion arises, $\varphi' \subseteq \varphi''$ stands for the pointwise ordering $\forall s \in S. \varphi'(s) \subseteq_T \varphi''(s)$.

Given $\varphi' : T \rightarrow U$ and $\varphi'' : S \rightarrow T$, the composition $\varphi' \circ \varphi'' : S \rightarrow U$ is defined as $\forall x. (\varphi'(\varphi''(x)) = \varphi'(\varphi''(x)))$. Composition preserves monotonicity ($\varphi' \in T_{\rightarrow} U$ and $\varphi'' \in S_{\rightarrow} T$ implies $\varphi' \circ \varphi'' \in S_{\rightarrow} U$), as well as continuity (the composition of two continuous functions is continuous) and isomorphism (composing two isomorphisms yields an isomorphism).

2.4.4 Fixpoints

A fixpoint $s \in S$ of $\varphi : S \rightarrow S$ is such that $\varphi(s) = s$. Fixpoints are elements of the set $\varphi^w = \{s \in S \mid \varphi(s) = s\}$. The least fixpoint LFP $(\varphi)$ is the unique $s$ such that $\forall t \in \varphi^w. t \sqsubseteq s$. The notion of greatest fixpoint GFP $(\varphi)$ is dual. By Tarski’s fixpoint theorem, if $\varphi$ is upper continuous on a complete lattice, the least fixpoint can be obtained (in no more than $\omega$ steps, where $\omega$ is the...
cardinality of natural numbers) as the limit of an ascending chain starting from the minimum:

\[ \text{LFP} (\varphi) = \sqcup_{n \geq 0} \varphi^n(\bot) \quad \text{where} \quad \varphi^0(s) = s \quad \text{and} \quad \varphi^{n+1}(s) = \varphi^n(\varphi(s)) \]

We write \( s = \text{LFP}_\bot (\varphi) \) to make clear that \( s \) is obtained as the limit of \( \{\varphi^n(\bot)\}_{n \geq 0} \).
3

Theoretical foundations and state of the art

It is a very sad thing that nowadays there is so little useless information.

Oscar Wilde (1854 - 1900)

People will accept your ideas much more readily if you tell them Benjamin Franklin said it first.

David H. Comins

This chapter introduces the main theories which are used throughout this work. Abstract Interpretation (Section 3.1) formalizes approximations on program data and provides a framework for program analysis. The security requirements we are going to investigate are described by the notion of Information Flow analysis (Section 3.2); the security property of Non-Interference is presented in its abstract version (Section 3.3). Next, an introduction to Program Slicing (Section 3.4) is included, because of its tight relation with Information Flow. Our work is directed to program certification, in the style of Proof-Carrying code, whose basic notions are introduced in Section 3.5. Finally, the state of the art is widely discussed in Section 3.6; the focus is mainly on current research in Information Flow analysis of Object-Oriented programming languages.

3.1 Abstract Interpretation

Abstract Interpretation was introduced by Patrick and Radhia Cousot in 1977 [19]. It is a theory of semantic approximation for systematically deriving semantic-based program analysis algorithms. Program analyzers rely on the
notion of *abstract property*, that is, a static approximation of the dynamic program properties.

Abstract Interpretation was refined, extended or generalized in several papers: (i) systematic derivation of program analysis frameworks [20, 22, 23, 25]; (ii) higher-order functional languages [24, 46]; (iii) logic languages [21]; (iv) types and type systems [18]. Further research steps on abstract domains were proposed by Filé, Giacobazzi, Mastroeni, Quintarelli, Ranzato and Scozzari in a series of papers [35, 36, 38, 37, 39, 34, 30, 33] (see [27] for a brief survey).

### 3.1.1 Program analysis frameworks

Abstract Interpretation is intended as a theoretical framework which allows to express a variety of program analyses into a unified formalism. Its main utilization is to design automatic program analyzers for statically determining dynamic program properties. This is the main goal of *Static Analysis*, that is, a class of techniques which aim to study programs without executing them. Abstract Interpretation defines static analyses, which can be compared or systematically modified/refined by means of the mathematical tools underlying the theory.

Static analysis should be decidable and efficient. The well-known, early foundational works in computer science by Turing, Kleene, Church, Post, Gödel and others, showed that most relevant program properties cannot be exactly decided by another program. The most famous example is the *halting problem*: no algorithm can tell, for every program $P$, whether the execution of $P$ will terminate. *Undecidable* properties often represent the most interesting information we want to acquire about a program. Static analyzers are forced to compute approximated properties with a soundness constraint; that is, the question *does the program $P$ have the property $\pi$?* cannot be affirmatively answered unless $\pi$ is indeed satisfied at the semantic, unapproximated level. Due to approximation, the inverse direction can be false, that is, it is possible that $P$ satisfies $\pi$ even if the analyzer cannot tell. Abstract interpretation formalizes the approximation of properties, and specifies methods to derive static analysis algorithms.

### 3.1.2 Principles of Abstract Interpretation

The classical framework for Abstract Interpretation starts from a semantics describing relevant aspects of program behavior. Several semantics [33] (operational, denotational, traces...) can be chosen, depending on the program properties we are interested in.

Semantics of Turing-equivalent programming languages are undecidable; usually, fixpoints (Section 2.4) describe the behavior of potentially nonterminating programs. Abstract Interpretation provides means for approximating a concrete fixpoint semantics into an *abstract semantics*. This technique can be seen as the simplification of the equation involved in the concrete semantics into an approximated abstract equation.
Approximation of concrete program properties

We assume concrete program properties to be described by elements of a given domain \( P \), equipped with an ordering relation \( \preceq \) such that \( \pi_1 \) is more precise than \( \pi_2 \) if \( \pi_1 \preceq \pi_2 \).

The notion *more precise* can be translated as *is satisfied by a smaller set of values* (which can be programs, program data etc.). Analogous definitions hold for abstract properties \( P^\sharp \).

Example. [Signs] Let \( P \) be the set \{\( \bot \), \( \neg \), \( 0 \), \( \text{mos} \), \( \text{nonPos} \), \( \text{non0} \), \( \text{nonNeg} \), \( \top \)\} which is intended as the value of some numerical variable \( x \) in a program. Therefore, the meaning of properties is \( \neg = 0 \), \( 0 = f0 \) etc., that is, a property is the set of values satisfying it, so that \( \neg \preceq \varnothing \). A possible approximation of \( P \) could be the domain \( P^\sharp = \{\bot^\sharp \), \( \text{nonPos}^\sharp \), \( 0^\sharp \), \( \text{nonNeg}^\sharp \), \( \top^\sharp \}\) with \( \text{nonPos}^\sharp = \text{nonPos} \) and \( \text{nonNeg}^\sharp = \text{nonNeg} \). The abstract set \( P^\sharp \) does not consider strict inequalities, then some loss of information occurs.

In many cases, when semantic values belong to a domain \( D \), the set of concrete properties is \( P = \varphi(D) \), that is, all possible properties of values are considered. The semantics of abstract properties is given by the concretization function \( \gamma : P^\sharp \rightarrow P \), which yields the concrete property corresponding to a given abstract property (for example, \( \gamma(\text{nonPos}^\sharp) = \text{nonPos} \)). The best abstract approximation of some property \( \pi^\sharp \) is given by the abstraction function \( \alpha : P \rightarrow P^\sharp \), where \( \alpha \) formalizes the idea of approximation.

Example. [Signs, continued] The abstraction and concretization functions for the previous example are shown in Figure 3.1.

The soundness of approximations, that is, the fact that \( \pi^\sharp \) is a valid approximation of \( \pi^\circ \), is expressed as \( \alpha(\pi^\circ) \preceq \pi^\sharp \) or, equivalently, \( \pi^\circ \preceq \gamma(\pi^\sharp) \). When these two conditions are equivalent, \( \alpha \) and \( \gamma \) are said to form a Galois connection.

**Definition 3.1 (Galois connection).** Given two posets \( (P^\circ, \preceq^\circ) \) and \( (P^\sharp, \preceq^\sharp) \), a Galois connection is a pair of maps \( \alpha : P^\circ \rightarrow P^\sharp \) and \( \gamma : P^\sharp \rightarrow P^\circ \) satisfying

\[
\forall \pi^\circ \in P^\circ, \pi^\sharp \in P^\sharp. \quad \alpha(\pi^\circ) \preceq^\sharp \pi^\sharp \iff \pi^\circ \preceq^\circ \gamma(\pi^\sharp)
\]
In this case, we write
\[\langle P^0, \leq^0 \rangle \overset{\gamma}{\longrightarrow} \langle P^\delta, \leq^\delta \rangle\]

The functional compositions \(\gamma \circ \alpha\) and \(\alpha \circ \gamma\) are, respectively, extensive and reductive (Section 2.4):
\[
\forall \pi^b \in P^b. \pi^b \leq^b \gamma(\alpha(\pi^b)) \quad \forall \pi^\delta \in P^\delta. \quad \alpha(\gamma(\pi^\delta)) \leq^\delta \pi^\delta
\]

It follows that \(\alpha\) and \(\gamma\) are monotone. Moreover, in a Galois connection, one of the two functions induces the other (the dual formula also holds):
\[
\langle P^0, \leq^0 \rangle \overset{\gamma_1}{\longrightarrow} \langle P^\delta, \leq^\delta \rangle \land \langle P^0, \leq^0 \rangle \overset{\gamma_2}{\longrightarrow} \langle P^\delta, \leq^\delta \rangle \implies
\]
\[
(\alpha_1 = \alpha_2) \implies (\gamma_1 = \gamma_2)
\]
Galois connections define closure operators. An upper closure operator (uco) is a monotone, extensive and idempotent map; a lower closure operator (lco) is reductive instead of extensive.

\[ P \xrightarrow{\gamma \circ \alpha} P^2 \quad \Rightarrow \quad \alpha \circ \gamma \text{ is an lco, } \gamma \circ \alpha \text{ is a uco} \]

Upper closure operators \( \rho \in \text{uco}(P^b) \), for a given concrete domain \( P^b \), will be used in the rest of this work. The set \( \rho(P^b) = \{ \rho(p^i) \mid p^i \in P^b \} \) is the set of fixpoints of \( \rho \) (by idempotence). If \( P^b \) is a complete lattice, then \( \rho(P^b) \) is a Moore family; \( \rho(P^b) \) is isomorphic to \( P^b \).

The uco-based approach to approximation is equivalent to the Galois connection approach. The former does not need the existence of some domain of abstract properties \( P^a \) to be made explicit. Instead, all abstract computations can be defined on a subset of the domain of concrete properties. The domain of concrete properties is called the concrete domain; depending on the context, either \( P^b \) or \( \rho(P^b) \) will be denoted as the abstract domain. Moreover, an abstract domain will be often identified with its upper closure operator. The elements of an abstract domain are its abstract values.

Example. The set of abstract properties \( P^a \) of Example 3.1.2 can be described by the upper closure operator \( \rho \) (when describing the same property \( V^\dagger \) is written as \( V \)):

\[
\begin{align*}
\rho &: \wp(\mathbb{Z}) \rightarrow \wp(\mathbb{Z}) \\
\rho(\emptyset) &= \emptyset \\
\rho([0]) &= \{0\} \\
\rho(X) &= \{n \mid n \geq 0\} \approx \text{nonNeg} & \text{if } \forall x \in X, x \geq 0 \\
\rho(X) &= \{n \mid n \leq 0\} \approx \text{nonPos} & \text{if } \forall x \in X, x \leq 0 \\
\rho(X) &= \mathbb{Z} \approx \top & \text{otherwise}
\end{align*}
\]

3.1.3 Abstract domains

Given a concrete domain \( P^b \) which is a complete lattice, the set of upper closure operators (or, equivalently, of Galois connections) is also a complete lattice ordered with respect to precision: the smaller a uco, the smaller the loss of information induced by the abstraction process.

\[ \rho_1 \preceq \rho_2 \triangleq \forall p \in P^b. \rho_1(p) \leq \rho_2(p) \]

Some operators are defined, which systematically refine, simplify or combine abstractions: among them, reduced product, disjunctive completion, complementation. We only introduce the reduced product.
Definition 3.2 (Reduced product). Let $P_i^\mathbb{1}_i$, with $i \in I$, be abstract domains designed over the concrete domain $P^\mathbb{R}_i$ by using Galois connections or upper closure operators. The reduced product $\cap_i(P_i^\mathbb{R})$ is the more abstract among the domains which are more concrete than every $P_i^\mathbb{R}$. It is defined as the Moore closure (Section 2.4) of the set $\bigcup_i P_i^\mathbb{R}$ or, equivalently, by the upper closure operator

$$\rho(\pi) = \bigwedge_i \rho_i(\pi)$$

The reduced product of a set of domains is the simplest domain which incorporates all the relevant information of the $P_i^\mathbb{R}$.

Example. Let $P_1^\mathbb{R}$ be the abstract domain of signs ($P^\mathbb{S}_1$ in Example 3.1.2) and $P_2^\mathbb{R}$ be the parity domain $\{[\top], [\text{even}], [\text{odd}], [\bot]\}$, with the obvious meaning for abstract values. The reduced product $P_1^\mathbb{R} \cap P_2^\mathbb{R}$ captures both the sign and the parity of numbers:

$$P_1^\mathbb{R} \cap P_2^\mathbb{R} = \{ [\bot], [0], [\text{nonPosEven}], [\text{nonPosOdd}], [\text{nonNegEven}], [\text{nonNegOdd}],[\top] \}$$

3.1.4 Property transformers and completeness

Galois connections can be lifted from sets of properties to sets of monotone property transformers: if $P_A^\mathbb{R} \xrightarrow{\gamma_A} P_B^\mathbb{R}$ and $P_B^\mathbb{R} \xrightarrow{\delta_B} P_A^\mathbb{R}$ are Galois connections, then

$$P_A^\mathbb{R} \xrightarrow{m} P_B^\mathbb{R} \xrightarrow{\lambda_B \circ \gamma_B \circ \delta_B} P_A^\mathbb{R}$$

where the orderings $\preceq_A$ and $\preceq_B$ on functions are pointwise. The function $\gamma_B \circ \psi \circ \delta_B$ is said to be the best correct approximation of $\psi$, that is, the most precise function approximating $\psi$ with respect to $\delta_B$ and $\gamma_B$.

This lifting can be used to approximate functions, functionals, etc. In general, the approximation must be sound, that is, for some $f^\mathbb{R} \in P_A^\mathbb{R} \xrightarrow{m} P_B^\mathbb{R}$ and $f^\mathbb{R} \in P_A^\mathbb{R} \xrightarrow{m} P_B^\mathbb{R}$, the following condition holds:

$$\alpha_B \circ f^\mathbb{R} \preceq_B f^\mathbb{R} \circ \alpha_A$$

Soundness implies that the result of applying the abstract function is a correct approximation of the concrete result. If equality holds, then we have completeness. Completeness is an important property of abstractions, meaning that, relatively to the underlying abstract domains, the abstract function is as precise as possible.
Example. The sign abstraction (Example 3.1.2) of properties $\wp(\mathbb{Z})$ is not complete with respect to the sum: the Galois connection is defined as $\wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \xrightarrow{m} \wp(\mathbb{Z}) \xrightarrow{\lambda \wp \circ \wp \circ \alpha} \text{SIGNS} \times \text{SIGNS} \xrightarrow{m} \text{SIGNS}$

where $\alpha$ and $\gamma$ induce the Galois connection $\wp(\mathbb{Z}) \xrightarrow{\gamma} \text{SIGNS}$. The sum function $f$ and its approximation $f^\dagger = \alpha \circ f \circ \gamma$ do not satisfy the completeness condition:

$$\alpha(f(3, -2)) = [\text{nonNeg}] \neq [\top] = f^\dagger(\alpha(3), \alpha(-2))$$

On the other hand, it turns out that this abstraction is complete with respect to product.

3.1.5 Fixpoint approximation

Let $f \in P^\rightarrow \mapsto P^\rightarrow$ be a continuous function, and assume we are interested in approximating its fixpoint $\text{LFP}_{\rightarrow}(f)$ (Section 2.4), obtained as the limit of the iteration sequence

$$f_0 = \bot^\rightarrow, \quad f_{n+1} = f(f_n) \quad \text{LFP}_{\rightarrow}(f) = \bigsqcup_{n \geq 0} f_n$$

It is natural to obtain the approximation by computing the abstract image of the sequence $\alpha(f_0), \alpha(f_1), \alpha(f_2), \ldots$. If the approximated function $f^\dagger$ is complete with respect to $f$ (that is, $\alpha \circ f = f^\dagger \circ \alpha$), then the fixpoint abstraction $\alpha(\text{LFP}_{\rightarrow}(f))$ can be obtained by the sequence

$$f_0^\dagger = \alpha(\bot^\rightarrow), \quad f_{n+1}^\dagger = f^\dagger(f_n) \quad \text{LFP}_{\alpha(L^\rightarrow)}(f^\dagger) = \bigsqcup_{n \geq 0} f_n^\dagger$$

If $f^\dagger$ is not complete, then the limit of the abstract sequence can be greater than $\alpha(\text{LFP}_{\rightarrow}(f))$.

Widening

Computing the limit of the abstract iteration sequence is not guaranteed to terminate. Widening operators [23] are used in computing abstract iteration sequences; they enforce the condition that every ascending sequence reaches a fixpoint in a finite number of steps.

Definition 3.3. Given a domain $L$ in which there exist infinite ascending chains (Section 2.4), the operator $\triangledown : L \times L \mapsto L$ is a widening if it satisfies
Theoretical foundations and state of the art

\[ \forall x, y \in L, \ x \sqsubseteq x \sqcup y \]
\[ \forall x, y \in L, \ y \sqsubseteq x \sqcup y \]

for all ascending chains \( x^0 \sqsubseteq x^1 \sqsubseteq \ldots \), the ascending chain defined by \( y^0 = x^0 \), \( y^{n+1} = y^n \sqcup x^{n+1} \) is not strictly ascending.

Applying \( \lor \) to an infinite strictly ascending chain (that is, such that \( \forall n. \ x^n \sqsubseteq n^{n+1} \)) builds another chain whose fixpoint is finitely reachable (that is, there exists an \( n \) such that \( y^n = y^{n+1} \)).

In the following, we will consider a bounded iteration, in which a maximum number of steps \( n \) is fixed. Unless the fixpoint has been reached before \( n \) steps, the \( n \)-th iteration is ulteriorly approximated in order to guarantee soundness.

3.1.6 Program properties

Semantics is the more concrete among the properties of a program. A program semantics identifies a set of programs which are semantically equivalent. Depending on how much the semantics is detailed [17] (for example, trace semantics is more detailed than input-output semantics since it can distinguish programs by their partial computations, not only by their final result), the set of semantically equivalent programs can be larger or smaller.

Given a semantics, \( \llbracket \cdot \rrbracket \), the collecting semantics \( \llbracket \cdot \rrbracket^C \) is a set extension of \( \llbracket \cdot \rrbracket \):

\[ \llbracket \mathcal{P} \rrbracket^C = \{ \llbracket \mathcal{P} \rrbracket \} \]

The collecting semantic can be seen as a very precise property of programs. In program analysis, we can be interested in less precise (but computable) properties; for example, requiring that the variable \( x \) at program point \( p \) has positive value identifies a set of program semantics which can be non-equivalent with respect to other properties.

Let \( P^o = \varphi (\{ \llbracket \mathcal{P} \rrbracket^C \}_{\mathcal{P}}) \) be the concrete domain of program semantics. An abstract semantics is an approximation of the collecting semantics. Given a set of abstract properties \( P^A \) and a program \( \mathcal{P} \), \( \llbracket \mathcal{P} \rrbracket^A \) is the most precise abstract property which is satisfied by \( \mathcal{P} \). Soundness implies that \( \llbracket \mathcal{P} \rrbracket^A \) must be greater than or equal to \( \alpha (\llbracket \mathcal{P} \rrbracket^C) \). In general, given an algorithm for computing the abstract semantics, it can be that the computed abstract property is less precise than \( \llbracket \mathcal{P} \rrbracket^A \); this is generally due to incompleteness, that is, some loss of precision induced by abstract computations.

Example. Let \( P^A \) be the set \{ \( \bot \), \( \text{pos} \), \( \text{nonPos} \), \( \top \) \}. Every program \( \mathcal{P} \) with positive final value for \( x \) will be classified in the same class with respect to \( P^A \).

Type systems are an interesting example of abstract semantics [18]: the values of data are abstracted into types, so that two concrete values are indistinguishable as long as they have the same type.
3.1.7 Typical (numerical) domains

We define here some abstract domains which will be useful in the following. If not differently specified, the concrete domain is intended to be $\mathbb{C} = \varnothing(\mathbb{Z})$. We will mainly refer to upper closure operators rather than Galois connections. Abstract domains are usually denoted by $\mathcal{A}$; when no confusion arises, $\rho$ will also stand for both the uco and the set of its fixpoints (equivalently, abstract properties or abstract values), that is, the corresponding abstract domain.

For an abstract value $V$, we write $V = X$ or $V \Downarrow X$, where $X \subseteq \mathbb{C}$, if $X$ is the set of concrete values which are described by $V$ (in terms of Galois connections, $X = \gamma(V)$). Moreover, $V = V' \cup V''$ is the value such that $\gamma(V) = \gamma(V') \cup \gamma(V'')$.

Numerical domains of interest are:

- **Sign, zero excluded:**
  \[
  \mathcal{A}_s = \rho_s = \{\top, [\text{pos}], [\text{neg}], \bot\}
  \]
  where $[\top] = \mathbb{Z}$, $[\bot] = \emptyset$, $[\text{pos}] = \{n \mid n \geq 0\}$ and $[\text{neg}] = \{n \mid n < 0\}$.

- **Sign, zero included:**
  \[
  \mathcal{A}_{sz} = \rho_{sz} = \{\top, [\text{pos}], [\text{neg}], [0], \bot\}
  \]
  where $[\text{pos}] = \{n \mid n > 0\}$ and $[0] = \{0\}$ (the other values being equal to $\mathcal{A}_s$). This domain might also include $[\text{nonPos}] = [\text{neg}] \cup [0]$ and $[\text{nonNeg}] = [\text{pos}] \cup [0]$.

- **Parity:**
  \[
  \mathcal{A}_p = \rho_p = \{\top, [\text{even}], [\text{odd}], \bot\}
  \]
  where $[\text{even}] = \{n \mid n \text{ is even}\}$ and $[\text{odd}] = \{n \mid n \text{ is odd}\}$.

- **Parity and Sign:**
  \[
  \mathcal{A}_{ps} = \rho_{ps} = \{\top, [\text{even}], [\text{odd}], [\text{pos}], [\text{neg}],\}
  \]
  $[\text{poseven}] = [\text{pos}] \cap [\text{even}]$, $[\text{posodd}]$, $[\text{negeven}]$, $[\text{negodd}], [\bot]\}
  \]
  which is the reduced product of $\mathcal{A}_s$ and $\mathcal{A}_p$.

The partial ordering on abstract values is illustrated in Figure 3.2.

3.2 Information Flow Analysis

The problem of protecting the confidentiality of information manipulated by computing systems is becoming more and more important [67]. The current practice of software production gives little assurance that programs satisfy constraints on how data are released (that is, how information flows). One of the most promising approaches to Information Flow security is based on the
use of programming-language techniques for specifying and enforcing security policies (*Language-based security*). Language-based mechanisms are especially interesting because of the inadequacy of standard security techniques in protecting confidential information in large, networked information systems such as the growing realm of web-based services.

Conventional security mechanisms, such as Access Control [48], require that agents have some privileges in order to access protected data. However, in a large computing system, assuming that all programs are trustworthy is simply not realistic: we are not guaranteed that checked programs (that is, which can indeed exhibit any certificate) will not leak information when manipulating confidential data. Attention must also be paid to how data propagate, in order to avoid that secret data become visible (by error or malice) to
3.2 Information Flow Analysis

untrusted users. The language-based approach to these security requirements relies on program analysis. Analysis should guarantee that information controlled by a confidentiality policy will not flow to locations which violate the policy (that is, which can be visible to unauthorized users). The main security property with respect to Information Flow is Non-Interference, introduced by Goguen and Meseguer [40] as:

One group of users [...] is noninterfering with another group of users if what the first group does [...] has no effect on what the second group of users can see.

Non-Interference relies on the separation of program data into private (or high-security, H) and public (or low-security, L). Public data are those which are directly visible by external agents. On the other hand, the value of private data should not be acquired, directly or indirectly, by untrusted, possibly malicious users. The usual framework underlying many formal definitions of Non-Interference identifies data with program variables: a program preserves confidentiality if, by observing the value of public variables after the program execution, an attacker is not able to acquire information about the initial value of private variables.

A formal definition of Non-Interference in IMP is:

\[
\forall h', h'' \in V^H, \forall l \in V^L. \quad ([\mathcal{P}] (h', l))^L = ([\mathcal{P}] (h'', l))^L
\]

where \( h \) and \( l \) are values for, respectively, the private variable vector \( h \) and the public variable vector \( l \): \([\mathcal{P}] (v_h, v_l)\) is the semantics of a program \( \mathcal{P} \) for some private and public input \( v_h \) and \( v_l \); \([\mathcal{P}] (v_h, v_l))^L\) is its restriction to public variables (that is, to what attackers can see). This means that executing twice \( \mathcal{P} \) with different values for the private input results in two final states which cannot be distinguished by external, untrusted users. Then, attackers are not able to infer the value of \( h \) by watching the final value of \( l \).

Example. Let \( \mathcal{P} \) be the IMP program below.

\[
11 = 1; \\
\textbf{while} (h1 > 0) \textbf{do} \{ \\
11 = 11 + 1; \\
h1 = h1 - 1; \\
\} \\
12 = 11 + h2;
\]

\( \mathcal{P} \) has two private variables \( h1 \) and \( h2 \), and two public variables \( l1 \) and \( l2 \). Both low-security variables have, after the execution, values which clearly depend on private input values. There is a direct information flow from \( h2 \) to \( l2 \), by means of the variable assignment in the last statement. Moreover, the final value of \( l1 \) clearly depends on \( h1 \) even if no forbidden assignment to the public variable occurs. This is an indirect form of information flow, in
which confidential information is leaked to public variables whose modification depends on private data. In this example, whether the increment of \( l_1 \) is executed depends on \( h_1 \). Analysis should also pay attention to transitive flows: \( l_2 \) depends on \( h_1 \) because it depends on \( l_1 \).

Studying the information flow of programs is essentially equivalent to computing dependencies between data [14, 2]: non-interference implies that public data do not depend on private data. One of the most practiced techniques for information flow analysis is the use of type systems [71, 70, 1]: a program satisfies the security requirements if it is typable with respect to some security type definition. Program analysis for Non-Interference has been applied to programming languages such as, among others, Java [54, 56, 6], Java bytecode [11], lambda-calculus [43] and ML [63].

3.3 Abstract Non-Interference

This section describes security properties obtained by weakening the notion of Non-Interference in order to model realistic confidentiality policies.

3.3.1 Weakening Non-Interference

The standard notion of Non-Interference turns out to be too strong a requirement for software production. Keeping public and private data completely separate can be unnatural and lead to limitations in the design of programs. In practice, some selected information leaking should often be allowed in real programs.

Example. If the access to some functionalities of the program \( P \) is regulated by password checking, the value of the password is clearly confidential. However, by trying to access the service, an external user acquires information about the password, namely, that it is different from his or her attempts. Such a program would not satisfy the standard property of Non-Interference; yet, it is perfectly reasonable.

Weakened forms of Non-Interference aim to specify which information flows from high-security to low-security data should be allowed. Many attempts have been proposed in this direction; among them, Declassification [75] and Relaxed Non-Interference [51].

3.3.2 Attackers as abstract interpretations

Non-Interference can be weakened by assuming that some dangerous flows are permitted since they cannot be exploited by attackers. That is, external,
unauthorized observers can only see public data up to a certain degree of precision. This is the assumption underlying Abstract Non-Interference (ANI) [31], a confidentiality policy introduced by Giacobazzi and Mastroeni in 2004. Abstract Non-Interference is based on Abstract Interpretation; it is parameterized on the precision (that is, the observational power) of attackers in observing the program execution. It is natural, in the Abstract Interpretation framework, to describe the observational power by means of the approximation induced by Galois connections. Attackers have access to an approximated version of the program semantics, that is, they can be described as abstract interpretation of programs.

**Example.** Let \( l \) be a public integer variable of the program \( P \). If an attacker \( A \) can observe the value of \( l \) before and after the execution of \( P \), then he or she has the maximum observational power on the datum. On the other hand, if \( A \) can only observe some property \( \pi \) of \( l \), then two executions in which \( l \) has the same value with respect to \( \pi \) (although, possibly, different concrete values) cannot be distinguished. Since a property on data can be represented by an upper closure operator \( \rho \), we say that the precision of \( A \) with respect to \( l \) is \( \rho \); the attacker cannot distinguish two values \( v_1 \) and \( v_2 \) if \( \rho(v_1) = \rho(v_2) \) (that is, if they are represented by the same abstract value or, equivalently, have the same property \( \pi \)).

In general, the observational power of attackers can be different for every public variable, and different between input and output. The following formal definition is the condition for a program to satisfy Abstract Non-Interference with respect to observational power \( \eta \) on the input and \( \rho \) on the output (with the same precision for all public variables).

**Definition 3.4 (Abstract Non-Interference).** A program is secure for ANI with respect to \( \eta \) and \( \rho \) (written \([\eta, \rho] P\)) if

\[
\forall h', h'' \in \mathcal{V}^H, \forall l', l'' \in \mathcal{V}^L. \quad \eta(l') = \eta(l'') \implies \rho\left(\mathcal{F}(h', l')^L\right) = \rho\left(\mathcal{F}(h'', l'')^L\right)
\]

If attackers cannot distinguish the public input, it is impossible for them to acquire information about private data by observing the output.

**Example.** Let the observational power of an attacker \( A \) be described by \( \rho_{sz} \). Then, the program \( P \) of Example 3.2 satisfies ANI for the variable \( l_1 \). In facts, the sign of \( l_1 \) is always positive and, therefore, does not depend on private data. On the other hand, an illicit flow is still possible with respect to \( l_2 \), since no hypotheses can be made on the initial value of \( h_2 \). Consequently, \( P \) is to be rejected even for the weakened security property; yet, some flows are no longer detected as dangerous.
3.3.3 Narrow and Abstract Non-Interference

Definition 3.4 allows some information flows to be detected as dangerous even if they are motivated by a change in public instead of private data. That is, it is possible that a flow from public data is considered as coming from private variables; these flows are named *deceptive flows*.  

*Example.* Let $l$ be the only public variable in $P$. Suppose an attacker can observe the parity $p$ of $l$ on the input and its sign $s$ on the output. Suppose also $l$ is not modified inside $P$; then, it is clear that no forbidden information flows should be detected. Yet, the ANI condition, as defined in Definition 3.4, does not hold since $\rho_p(-4) = \rho_p(6)$ but $\rho_s(-4) \neq \rho_s(6)$. 

To deal with this problem, a new definition of Non-Interference must be provided: namely, the *Abstract Non-Interference* [31] of Definition 3.4.

**Definition 3.5.** A program is secure (written $(\eta) P (\phi \rightsquigarrow [\rho])$) if, for every $h', h'' \in \mathcal{V}^\Pi$, $l', l'' \in \mathcal{V}^L$:

$$\eta(l') = \eta(l'') \implies \rho \left( ([P] (\phi(h'), \eta(l')))^L \right) = \rho \left( ([P] (\phi(h''), \eta(l'')))^L \right)$$

The semantics of $P$ is computed on abstract values with $\phi$ and $\eta$.

The idea is to model only information flows generated by the variation of private values. This is obtained by computing the program semantics on abstract values. This refined definition of ANI has no deceptive flows: if an attacker can observe the property $\eta$ of public input and $\rho$ of public output, then no information flow concerning the property $\phi$ of private input interferes in the observable property $\rho$, under the assumption that the public input property $\eta$ does not change. The following holds:

$$[ID] P (ID) \implies [\eta] P (\rho) \implies (\eta) P (ID \rightsquigarrow [\rho]) \implies (\eta) P (\phi \rightsquigarrow [\rho])$$

where $ID = \lambda \pi. \pi$ is the *identity domain*. The standard Non-Interference property can be expressed as $[ID] P (ID)$, or $(ID) P (ID \rightsquigarrow [ID])$.

### 3.4 Program Slicing

Program slicing [72, 45] is a software engineering technique which has been successfully used in many areas as debugging, testing, reverse engineering, reuse and complexity analysis. Its main goal is to get a deeper understanding of a program by finding, for a given behavior (the *slicing criterion*), the subset of the program (the *slice*) which is relevant to that behavior. Program slicing relies on the notion of dependency between program data: $a$ depends on $b$ if its value can change due to some changes in $b$ (for dependencies, see Sections
3.4 Program Slicing

3.6.2 and 5.2). A statement is not relevant for a criterion if what the statement does has not influence on the behavior which is induced by the criterion.

Slicing comes in different versions: static slicing does not make assumptions about the input of a program. On the other hand, dynamic slices [47] are computed by relying on a specific test case, and are usually smaller. Another variant is conditioned slicing [13], in which an input condition represents a set of initial states; a theorem prover extracts the statements which are never executed if the condition induced by the slicing criterion is satisfied.

3.4.1 An algorithm for program slicing

The literature on program slicing is well-known (see [69] for a survey). Here, we illustrate a basic, non-interprocedural algorithm for static slicing [72] of IMP programs, based on the iterative solution of data flow equations.

A slice is a correct program which is obtained from the original program by deleting zero or more statements, namely, those which are not relevant to a particular criterion. A slicing criterion is a pair \((s, v)\) where \(s\) is a program point (or the program statement or control predicate at that point) and \(v\) is an identifier. A slice for a criterion \((s, v)\) must compute, for \(v\), the same value as the original program when the program is at \(s\). In the following, sets \(S\) of slicing criteria will be considered.

The algorithm relies on dependencies between statements, which can be data dependencies (occurring when a statements modifies or defines an identifier which is referred to by another statement) or control dependencies (a statement is control dependent on a control predicate if the predicate has influence on whether the statement is executed.).

A control flow graph CFG is computed for the program. Nodes are program statements (identified by numbers), edges \(i \rightarrow_{\text{cfg}} j\) represent the dependency of statement \(j\) on \(i\). The functions \(\text{def}(\cdot)\) and \(\text{ref}(\cdot)\) are defined as

\[
\text{def}(v := e) = \{v\}
\]

\[
\text{ref}(v := e) = \text{ref}(\text{if } e \text{ then } \cdot \text{ else } \cdot) = \text{ref}(\text{while } e \text{ do } \cdot) = \text{vars}(e)
\]

Let \(S\) be a set of slicing criteria, that is, \(S = \{(k, v) \mid v \in V\}\) for a statement \(k\) and some set of identifiers \(V\). This means that we are interested in the value of identifiers in \(V\) at statements \(k\). Then, the following dataflow equations are defined: for each \(i \rightarrow_{\text{cfg}} j\),

\[
R_S^0(i) = R_S^0(j) \cup \{v \mid v \in R_S^0(j) \land v \notin \text{def}(i)\}
\]

\[
\cup \{v \mid v \in \text{ref}(i) \land \text{def}(i) \cap R_S^0(j) \neq \emptyset\}
\]

\[
S_S^0 = \{i \mid i \rightarrow_{\text{cfg}} j \land \text{def}(i) \cap R_S^0(j) \neq \emptyset\}
\]

The first formula is the set of directly relevant variables; the second is derived from the first and is the set of relevant statements. Iteration starts with \(R_S^0(n) = \forall\) and \(R_S^0(m) = \emptyset\) for \(m \neq n\), where \(n\) is the end node of CFG.
Then, control dependencies are taken into account, that is, those originating from control statements. Let $\text{INFL}(b)$ the set of statements which are under the range of influence of a predicate $b$. Dependencies are propagated as follows:

$$B^k_S = \{ b \mid S^k_S \cap \text{INFL}(b) \neq \emptyset \}$$

$$R^{k+1}_S (i) = R^k_S (i) \cup \bigcup_{b \in B^k_S} R^0_{\{(b,v)\mid v \in \text{REF}(b)\}} (i)$$

$$S^{k+1}_S = B^k_S \cup \{ i \mid i \rightarrow_{\text{CFG}} j \land \text{DEF}(i) \cap R^{k+1}_S (j) \neq \emptyset \}$$

$R^{k+1}_S (i)$ is obtained by adding to $R^k_S (i)$ variables which are relevant because they have a transitive data dependency on statements in $B^k_S$ (that is, predicates which have influence on statements in $S^k_S$).

The set $S^k_S$ is non-decreasing on $k$; the fixpoint of the iterative computation is the desired program slice.

### 3.4.2 Slicing and information flow

An Object-Oriented approach to slicing can be found in [49] and [50]; the second paper highlights the link between slicing and information flow. The relation of slicing with information flow was first outlined by Bergeretti and Carré [12].

Let $\mathcal{P}'$ be a program which is obtained by another program $\mathcal{P}$ by adding, for every private variable $h$, a statement $i_h : h := h$ at the beginning of $\mathcal{P}$ ($i_h$ is the corresponding program point). These statements play the role of input statements. Similarly, for every public variable $l$, the statement $i_l : l := l$ is added at the end of $\mathcal{P}$ (representing the output of $l$). We have that $\mathcal{P}$ satisfies the Non-Interference condition if the slice of $\mathcal{P}'$ with respect to the slicing criterion $\{(i_h, l) \mid l \text{ is public} \}$ does not contain any of the $i_h$ statements, that is, if the initial value of the private variables does not affect the final value of public variables.

**Example.** The IMP program $\mathcal{P}$, with private variables $h_1, h_2$ and public variables $l_1, l_2$, and its transformation $\mathcal{P}'$ are shown in Figure 3.3.

If the slice is computed with respect to the public values at the last two assignments, then it does not contain $h_2 := h_2$. However, since it does contain $h_1 := h_1$, Non-Interference is not satisfied. In facts, the public variable $l_1$ indeed depends on the initial value of $h_1$. $\square$

The relation between Information Flow and Program Slicing will be also discussed (with respect to the abstract notion) in Section 5.6.
3.5 Proof-Carrying Code

The Proof-Carrying code (PCC) framework was proposed by George Necula in 1997 [58, 57, 59, 60, 61]. Through the program certification mechanism provided by a Proof-Carrying code architecture, a host system is able to determine whether it is safe to execute a program which is supplied by an untrusted source.

With the exponential growth of Web-based communication, it happens more and more frequently that a user be asked to execute untrusted code. Certification should be provided in order to guarantee security properties of programs. The mechanism of certificates (that is, a program is executed only if the source can be trusted) seems not to be sufficient, mainly due to the impossibility to maintain a correct knowledge of trusted sources. Proof-Carrying code follows a different path: programs are executed if it is possible, for the user, to check safety with respect to some agreed security policy.

3.5.1 The PCC architecture

In Proof-Carrying code, the code producer is required to provide a safety proof for the program with respect to some security policy. The safety proof comes along with the program (if the producer and the user agree on the security policy), or is provided on-demand to prove that the program satisfies the required security properties. In any case, before executing the code, the consumer runs a proof validator to determine the validity of the provided proof; if the checks is successful, the program can be safely executed.
Basically, there are no theoretical constraints on which methods should be used in order to obtain, represent and validate the safety proof. The key point is that validating a proof (which is on the side of the consumer) is, in general, a much cheaper task than obtaining it (which is on side of the producer). An overview of the PCC architecture is shown in Figure 3.4. The part of the diagram which is beyond the dotted line is the trusted component of the PCC architecture; that is, the only module the consumer is forced to consider as safe without having a proof. The key advantage of Proof-Carrying code relies on the simplicity and the efficiency of a proof validator, compared to the complexity of a theorem prover; the hard work is left to the code producer, which can use various optimizing techniques in order to boost the program analysis. The verification condition (VC) is a predicate, expressed in some logic, whose validity is a sufficient condition for ensuring compliance with the safety policy. The code producer computes VC for proving it, while the consumer computes it to ensure that the code contains a valid proof.

**Fig. 3.4.** The Proof-Carrying code architecture

3.5.2 A novel approach to PCC architecture design

Recently, a new design for a PCC architecture has been proposed by Chang, Chlipala and Necula [15]. Instead of a safety proof, a program is equipped with a static analyzer for the security property and an analysis certifier, whose soundness must be checked by some trusted algorithm. The code user runs the analyzer on the program and checks the correctness of the analysis. According
3.5 Proof-Carrying Code

to the authors, this should lead to several benefits in modularity and flexibility; moreover, the analysis certifier is expected to be easier to prove sound than the analysis itself.

3.5.3 Application examples

Extension of the ML runtime system

In the original paper [58] an example is provided, in which Proof-Carrying code allows arbitrary users to use untrusted, foreign functions in the runtime system of a safe programming language. In practice, when programming in high-level languages such as ML or Haskell, the runtime system must be extended with functionalities which are written in an unsafe programming language, such as C or even assembly language. Here, the problem is to guarantee that foreign operations preserve some conditions on data representation. The following ML code defines a union type \( T \) and a sum operation, which computes the sum of all numbers contained in the \( T \) list. Suppose that the runtime system uses, for sum, a hand-optimized assembly version. Then the problem of defining a safety policy for ML runtime system extensions arises; the foreign sum function must be proven correct with respect to the safety policy.

```ml
datatype T = Int of int | Pair of int * int
fun sum (l : T list) =
  let
    fun foldr f nil a = a
    | foldr f (h :: t) a = foldr f t (f (a, h))
  in
    foldr (fn (acc, Int i) => acc + i
      | (acc, Pair (i, j)) => acc + i + j)
    l 0
  end
```

The safety requirement for runtime system extensions is that they preserve data-representation conventions. In ML, data representation is type-directed and can be expressed for this example as follows:

\[
\tau ::= \text{int} | \tau_1 \times \tau_2 | \tau_1 + \tau_2 | \text{list}
\]

\( T \) represents the type \( \text{int} + (\text{int} \times \text{int}) \). Conditions on data-representation are expressed in terms of type safety. Integer values are \( k \)-bit machine words, depending on the chosen architecture; pairs are represented as pointers to a pair of location containing the constructor value (0 for Int, 1 for Pair) and the value contained in the constructor.

A verification condition is computed in Floyd style by a VC generator; it is based on the precondition and the postcondition for program safety:
where $r_0$ is the machine register which contains the input list at the beginning, and the final result at the end of the execution. The VC generator uses pre and post to generate a formula which is a sufficient condition for the preservation of data representation invariants. The assembly code of the sum function is annotated with invariants, that is, conditions to be satisfied at a given program point: for example,

\[
\text{INV } r_0 : \text{T list}
\]
is the precondition before the first code instruction, while

\[
\text{INV } r_0 : \text{T list} \land R_1 : \text{int}
\]

must hold at every loop iteration ($r_1$ contains the value of acc). The soundness of the VC generator must be proven; that is, given a pre and a postcondition and a set of invariants, the validity of VC implies that the program reads only from valid memory locations as they are defined by the typing rules.

The safety proof is a derivation of the Verification Condition. Edinburgh Logical Framework [42] was advocated as a formalism to represent predicates and proofs; however, any other reasonable framework can be used. Further details are omitted.

3.6 The state of the art: Object-Oriented Information Flow analysis

This section introduces some recent results in the field of Information Flow analysis, with a particular interest in Object-Oriented programming languages.

Section 3.6.1 largely discusses an approach to Non-Interference analysis in presence of pointers. This work deals with aliasing, in a more refined way than existing type-based approaches. Current research on abstract dependencies is illustrated in Section 3.6.2. The analysis of low-level languages, together with code certification and the preservation of properties through compilation, are discussed in Sections 3.6.3 and 3.6.4. A short survey of mathematical formalisms for describing information flow is given in Section 3.6.5. Finally, Section 3.6.6 introduces the only existent approach to ANI verification and outlines some of its shortcomings.

3.6.1 Analysis with pointer aliasing

A recent paper [4] by Amtoft, Bandhakavi and Banerjee addresses the problem of pointer aliasing [16, 26] in Information Flow analysis of Object-Oriented programs. Pointer aliasing occurs whenever two identifiers can refer to the
same memory location(s), as in languages with explicit pointers (C or C++) or Object-Oriented languages (C++ or Java).

In Java, an object of class C identifies a portion of memory which can be referred to by any identifier whose type is compatible with C. Consequently, any modification to the object via one of the identifiers referring to it propagates to the other identifiers. Program analysis must account for this problem, by specifying when it is safe to consider two identifiers x and y as **independent**, that is, when x and y are guaranteed not to share any memory locations.

**Assertions**

Independence properties are specified by means of assertions in a Hoare-like logic. The logic basically relies on two predicates: region assertions and independence assertions. The first has the form \( x \leadsto L \), where \( x \) is an identifier and \( L \) is an abstract location, that is, an abstract representation of a set of heap locations. Similarly, the assertion \( L_1.f \leadsto L_2 \) means that, for every concrete location \( l_1 \) which is abstracted by \( L_1 \), any location \( l_2 \) contained in the field \( f \) of \( l_1 \) is represented by \( L_2 \). If \( L_1 \) and \( L_2 \) are disjoint, then the assertions \( x \leadsto L_1 \) and \( y \leadsto L_2 \) imply that \( x \) and \( y \) cannot alias, since they do share no heap locations. This is an independence assertion, and can be expressed as \( x \not\equiv y \) (to be read as \( x \) is independent of \( y \)). Assertions and statements are combined in triples \( \{ \phi_i \} S \{ \phi_o \} \) where the assertions \( \phi_i \) and \( \phi_o \) are valid, respectively, before and after the statement \( S \). In addition to assertions which are computed axiomatically, programmers are allowed to insert programmer assertions are also, in order to help the prover to deal with independence results which are clear but not easily provable.

**Semantics of assertions**

States \( (s, h) \) contain a store \( s \) and a heap \( h \). The extraction relation \( \eta \) relates locations to abstract locations. It is, actually, an upper closure operator. The one-state semantics gives a precise meaning to assertions \( \phi \) by the predicate \( (s, h) \models_{\eta} \phi \) (\( l.f \) is the field \( f \) of the location \( l \)):

\[
(s, h) \models_{\eta} x \leadsto L \iff (s(x), L) \in \eta \\
(s, h) \models_{\eta} L.f \leadsto L' \iff \forall l. (l, L) \in \eta \Rightarrow (l.f, L') \in \eta
\]

A two-state semantics is also defined, which relates two states with an assertion. When there is no ambiguity on \( y \), the assertion \( x \not\equiv y \) stands for \( x \not\equiv y \). The meaning of \( x \not\equiv y \) is that the two states which are referred by the two-state semantics agree on the value of \( x \). As an example of semantic rule,

\[
(s, h) \& (s', h') \models_{\beta, \eta, \eta'} x \not\equiv (s(x), s'(x)) \in \beta
\]

means that \( x \) is constant in \( (s, h) \) and \( (s', h') \) if its location in the two states is related by a bijection \( \beta \) on locations.
Static independence checking

Dependencies are checked by means of a Hoare-like logic. A *judgement* of the logic takes the form

$$\Pi \vdash \{ \phi_i \} S \{ \phi_o \} [X]$$

where $X$ is an over-approximation of the abstract locations modified by $S$, $\Pi$ is an environment of method summaries $\{ \psi_i \} \vdash \{ \psi_o \} [X']$ which describe the behavior of methods, and $\{ \phi_i \} S \{ \phi_o \}$ is a Hoare triple. An *axiomatic semantics* is provided, which computes assertions by means of a set of rules (for example, $\{ y \rightarrow L \} x := y \{ x \rightarrow L \} [\{x\}]$ is the rule for variable assignment). Judgements of the logic only refer to what is actually affected by a statement; they are combined by means of a sound *frame rule*, in the style of *Separation Logic* [64].

A sound algorithm $\sigma \rho (S, \phi)$ computes, for a statement $S$ and a precondition $\phi$, a pair $(\phi_o, X)$ where $\phi_o$ is a postcondition and $X$ approximates the abstract addresses modified by $S$. Under certain conditions, $\phi_o$ is proven to be the *strongest postcondition*.

Discussion

This work uses Hoare logic to define an interprocedural, flow sensitive Information Flow analysis for Object-Oriented programs, extending a previous work [5] on a simple imperative language. The use of region and independence assertions allows a better treatment of aliasing than current type-based analyses [54, 7]. Besides describing Non-Interference issues, assertions also model aliasing.

Methods are represented by summaries, in form of triples. A method is assigned a set of summaries, since different preconditions and postconditions may hold at different call sites. This *polyvariance* does not, however, deal with subclasing and dynamical dispatch: these two issues are not taken into account in this work. Dealing with multiple method definitions would probably imply the necessity of disjunctions in assertions (actually, disjunctions are confined in programmer assertions).

A collection of rules for computing postconditions is shown. The precondition of

$$\{ x \rightarrow L, y \rightarrow L', L.f \rightarrow L'; x:\kappa, y:\kappa, L.f:\kappa \} x.f = y \{ L.f \rightarrow L'; L.f:\kappa \} [\{L.f\}]$$

specially in its $L.f \rightarrow L'$ component, seems to be a bit strong (assignment cannot change the abstract location), and satisfied only if abstract locations contain all type-compatible concrete locations. This could possibly result in limiting the benefits of this approach in the treatment of aliasing. Also, it is not clear how the assumption on the existence of a specific location $L_0$ for new objects restricts the set of programs which can be accounted for in the analysis.
3.6 The state of the art: Object-Oriented Information Flow analysis

3.6.2 Abstract dependencies

Abstract dependencies are accounted for in another recent work [65], where they are applied to the detection of false alarms in program analysis, and in the context of semantic slicing [66].

Classical, concrete dependencies (see Definition 5.5, and [45] for a calculus on dependence graphs) keep track of which data can have their value modified, in response of some changes in data they depend on. Given a program $P$, its trace semantics collects all the traces $(s_0, \ldots, s_n) \in \Sigma$, that is, the execution of $P$ on every possible input state $s_0$, ending in $s_n$. Standard dependencies can be generalized in two ways, as shown in the following paragraphs.

Observable dependencies

In program analysis, we are often not interested in all data dependencies. On the contrary, we want to identify a subset of the program traces and consider only dependencies which are observable in this subset. Given a semantic slice $E \subseteq \Sigma$ and a function $E^2$ mapping program points to sets of memory configurations, we have that $E^2$ is an abstraction of $E$ if the memory configuration $\mu \in M$ belongs to $E^2(l)$ for all traces $(\ldots, (l, \mu), \ldots) \in E$. Intuitively, a dependency is observable if it is a dependency on the whole set of traces, and it still exists in the restriction induced in the semantic slice. We are not interested in this restriction, then all traces will be considered in the following paragraph.

Abstract dependencies

The other restrictive\footnote{Restrictive in the sense that it identifies a smaller set of dependencies than the standard notion.} definition of dependency is modeled on an abstract property: the only dependencies which are taken into account are those modifying the abstract property of dependent data.

**Definition 3.6 (Abstract dependencies).** Let $\phi$ be a denotation, that is, a function mapping $M$ (memory configurations) to $\phi(M)$, and $\alpha_0$ and $\alpha_1$ be the abstraction functions (with the dual $\gamma_0$ and $\gamma_1$) of some abstract domains $D_0$ and $D_1$ on data. Then, $\phi$ induces an abstract dependency of $(x_1, \alpha_1)$ on $(x_0, \alpha_0)$ if and only if there exist a memory configuration $\mu$ and two abstract values $d_a, d_b \in D_0$ such that

$$\alpha_1(\phi(\mu [x_0 \leftarrow \gamma_0(d_a)])(x_1)) \neq \alpha_1(\phi(\mu [x_0 \leftarrow \gamma_0(d_b)])(x_1))$$

This definition (similar to Definition 5.6) means that there is an abstract dependency from $x_0$ to $x_1$ if changing the abstract property for $x_0$ in the input configuration of $\phi$ results in a different abstract value for $x_1$ in $\phi$ output (that is, after applying the denotation to the initial state).
Comparing and computing dependencies

The abstraction functions parameterizing the above definition underline a hierarchy of abstract dependencies: the more precise \( \alpha_0 \) and \( \alpha_1 \) are, the more dependencies will be discovered. Having \( \alpha_0 = \alpha_1 = \lambda v . v \) leads to standard dependencies.

Furthermore, definitions are provided, which show how it is possible to combine dependencies:

- Dependencies on the denotations \( \phi' \) (on \( \alpha_0 \) and \( \alpha_1 \)) and \( \phi'' \) (on \( \alpha_1 \) and \( \alpha_2 \)) can be composed into an upper approximation of the dependencies of \( \phi_2 \circ \phi_1 \) on \( \alpha_0 \) and \( \alpha_2 \).
- Given a set of paths from \( l_0 \) to \( l_n \), dependencies can be approximated by fixpoint iteration of local dependencies on all the traces from \( l_0 \) to \( l_n \).

Importantly, aliasing issues are not taken into account.

Discussion

This approach is interesting since it shows how to deal with the computation of abstract dependencies. It is claimed to be tightly connected with Non-Interference and its abstract version, ANI. Moreover, strong connections with program slicing are outlined.

Abstract dependency is an approximated property on program data. It is a weak version of the standard notion of dependency since the information \( a \) is said to be relevant to \( b \) only if a change in \( a \) can modify some abstract property (not only the concrete value) of \( b \). In the work of Rival, a number of techniques is provided for analyzing and composing dependencies, based on the trace semantics of programs. For example, the dependency analysis of the compound statement \([s_1 ; s_2] \) involves combining dependencies in \( s_1 \) and \( s_2 \) via some compositional rule. However, when combining dependencies, an assumption is made on the analysis of simple statements: only a mathematical, set-theoretic definition is provided for the dependency calculus. Then, we need to fill the gap between a purely semantic, mathematical notion and an algorithmic definition. To the best of our knowledge, there is no other work in the literature addressing this topic, which is among those discussed in Section 5.2. Clearly, in order to be decidable and as efficient as possible, many approximations are unavoidable, thus resulting in some loss of precision.

3.6.3 JVM-like programming languages

Barthe and Rezk [10, 11] propose an information flow type system for a non-trivial fragment of the Java Virtual Machine, including classes, methods and exceptions. The type system is developed in four layers:
3.6 The state of the art: Object-Oriented Information Flow analysis

1. operations for manipulating operand stacks and jumps [8]; every local variable has a security level; a notion of indistinguishability is defined. Instructions inside conditional branches can have their security level lifted by the boolean guard.

2. Classes, static fields and static methods (respectively playing the same rôle as global variables and procedures); methods are equipped with a security signature $k_1 \xrightarrow{k} k_2$, where $k_1$ and $k_2$ are, respectively, the expected security level of the parameters and the result, and $k$ is the effect of the method on the global memory.

3. Dynamically created objects, method overriding, type casts; here, the main complication is the use of a dynamically allocated heap.

4. Exceptions, which can leak information through abnormal termination.

Soundness proofs are provided for each level, based on the chosen notion of state indistinguishability.

Discussion

The proposed information flow analysis accounts for a large subset of Java bytecode (but the treatment of overriding is quite unclear). However, the type system seems to be overly conservative, particularly when using security signatures. Moreover, a common feature of type-based approach is the lack of flow-sensitivity: the existence of an unsafe subprogram is a sufficient condition for rejecting a program.

Remark 3.7. Basically, security levels for variables play the same rôle as boolean variables in Chapter 5. In order to account for more complex information flow properties, such as Abstract Non-Interference, the algorithm with boolean variables is equipped with mechanisms for computing relevant variables in expressions. Probably, this can also be applied to this (standard) information flow type-system, in order to analyze ANI.

3.6.4 Code certification

To the best of our knowledge, there does not exist a complete work on code certification for Information Flow security properties. Since certification deals with properties of executable programs, rather than source code programs\(^3\), a step in this direction can be found in a recent work by Barthe, Basu and Rezk [8].

Here, the focus is set on the preservation of security program properties through plans. A security type system for a low-level programming language with jumps and calls is provided, modeled on its counterpart on the high-level

\(^3\) In facts, verifying source code can help the code producer to build good programs, but is not a guarantee for the end user, since the program comes as executable.
language. This work introduces a compiler, together with a proof that compilation preserves security properties. The proof can be seen as a procedure to compute, from a well-typing certificate for the source program, another well-typing certificate for the compiled program.

3.6.5 Mathematical formalisms

Besides security type systems [71], several other techniques have been used in order to describe the information flow properties of programs. Among them, Hoare logic [5], self-composition with Hoare logic [9] (see below), boolean functions [28, 29], dependency calculi [14].

To the best of our knowledge, there does not exist a formal and exhaustive comparison between all or some of these techniques (for example, with respect to what they can prove, their efficiency, their flexibility etc.).

Self-composition

This technique relies on the following assumption: a program \( \mathcal{P} \) is secure if the sequential composition \( \mathcal{P}; \mathcal{P}' \) satisfies the Hoare triple

\[
\{ \forall x \in X_1.x = x' \} \quad \mathcal{P} \quad \{ \forall x \in X_1.x = x' \}
\]

where

- \( \mathcal{P}' \) is obtained by \( \mathcal{P} \) through variable renaming;
- \( X_1 \) is the set of public variables;
- for every \( x, x' \) is the renaming of \( x \) in \( \mathcal{P}' \).

It is easy to see that this definition is precisely Non-Interference in its original formulation [40].

Which Object-Oriented notion of Non-Interference

Existing literature on Information Flow analysis for realistic Object-Oriented programming languages often provides tiny examples of illicit flows (for example, a dangerous assignment of an instance field) in order to show how algorithms behave. However, a formal characterization of Non-Interference for a complete Object-Oriented program (in the case of Java, a set of cooperating class declarations with an entry point) cannot be found in the works referred in this section, with the exception of the work by Myers, Banerjee and Naumann [54, 55, 6]. The discussion of Section 4.3 is in this direction.
3.6.6 Proving Abstract Non-Interference

Giacobazzi and Mastroeni [32] propose an axiomatic proof system for Abstract Non-Interference, based on Hoare logic. Assertions (for narrow ANI) take the form $[\eta] s(\rho)$, where $s$ is a statement. Derivation rules allow to infer an assertion given one or more premises. Examples of the derivation rules are

\[
\frac{[\eta] s \top }{[\eta] s_1(\rho) \quad [\eta] s_2(\rho)}
\]

meaning, respectively, that (i) every statement satisfies ANI if no property can be distinguished at the output; (ii) the ANI property on $\eta$ and $\rho$ is preserved by sequential composition. There are rules for computing loop invariants, as well as non-structural rules for extending the obtained results (here, $\subseteq$ means more precise):

\[
\frac{[\eta'] s(\rho') \quad \eta \subseteq \eta' \quad \rho' \subseteq \rho}{[\eta] s(\rho)} \quad \forall i \in I. \quad [\eta] s(\rho_i)
\]

In this system, it is possible to prove some results about invariants and the security of programs with respect to given abstract properties. However, in the rule for public variable assignment ($\Pi(\eta)$ refers to partitions, and will be taken into account here)

\[
\frac{[\eta] e(\rho) \quad \Pi(\eta) \subseteq \Pi(\rho)}{[\eta] \left[ x := e(\rho) \right]}
\]

the assertion $[\eta] e(\rho)$ is defined as

\[
\forall l_1, l_2 \text{ public. } \eta(l_1) = \eta(l_2) \implies \forall h_1, h_2 \text{ private. } \rho(e(h_1, l_1)) = \rho(e(h_2, l_2))
\]

It can be easily seen that this is the set-theoretical definition 3.4 of narrow Abstract Non-Interference. Consequently, the main point in proving ANI has a mathematical, non-algorithmic formulation. Actually, this is the main difficulty in proving abstract information flow properties.
4 Representing Abstract Non-Interference

All the limitative Theorems of metamathematics and the theory of computation suggest that once the ability to represent your own structure has reached a certain critical point, that is the kiss of death: it guarantees that you can never represent yourself totally. Godel’s Incompleteness Theorem, Church’s Undecidability Theorem, Turing’s Halting Problem, Turski’s Truth Theorem—all have the flavour of some ancient fairy tale which warns you that ”To seek self-knowledge is to embark on a journey which... will always be incomplete, cannot be charted on a map, will never halt, cannot be described.”

Douglas R. Hofstadter

What a good thing Adam had. When he said a good thing he knew nobody had said it before.

Mark Twain (1835 - 1910), Notebooks

This chapter investigates how to represent the information flow behavior of programs with respect to abstract properties. Section 4.2 describes a type-based approach and its applications to simple languages. Next, we discuss the formalization of Abstract Non-Interference in a realistic Object-Oriented framework. Typical numerical domains are defined in Section 3.1.7.

4.1 Introduction

Checking the security property of Abstract Non-Interference relies on finding a suitable approach to the representation of abstract properties. This is tightly related to the representation of abstract domains and operations on abstract values. ANI is defined as a denotational property of input and output data
(Definitions 3.4 and 3.5); its set-theoretic flavour (made visible by the use of universal quantifiers) is a major obstacle towards the automatizing of analysis techniques.

In principle, if $D$ is an infinite set, there is no way to automatically express values and operations belonging to an arbitrary abstract domain on $D$. In particular, two operations are crucial: (i) computing the abstract property $\rho(v)$ of a concrete value $v$; (ii) combining two abstract values by means of some abstract operation $V' \otimes V''$.

**Values and operators**

Task (i) can be accomplished by some computable function $\rho()$, taking $v$ as its input and yielding a representation of the abstract values describing $v$. Yet, a Turing-equivalent programming language is needed in order to represent arbitrary domains\(^1\). The use of a universal language involves problems with termination, which become specially important when the process of code analysis is embedded into a certification architecture (Section 6.3): a user is probably not willing to validate programs by means of some algorithm which is not guaranteed to terminate. An extant approach to represent abstract properties via a computable $\rho()$ can be found in \cite{51}\(^2\); in this case, many domains cannot be described since the adopted language (a typed lambda-calculus without recursion) is not universal. Moreover, some computable domains cannot be easily represented in analytical form. Instead, tabular representation would be more appropriate, with the important limitation of being unable to deal with infinite abstract domains.

Task (ii) suffers from the similar problems, namely, our inability to deal with infinite sets. In order to avoid set theoretic operations such as $V' \otimes V'' = \alpha(\{v' \otimes v'' | v' \in \gamma(V'), v'' \in \gamma(V'')\})$, rules must be provided which specify the abstract behavior of $\otimes$. Importantly, partial knowledge about abstract operations preserves soundness, that is, when no rule $V' \otimes V'' = U$ is provided, the result can be safely approximated by $[\top]$ (thus inducing less precise computations).

### 4.2 Security types

This section illustrates an approach \cite{73, 74} to the representation of Abstract Non-Interference, which uses *security types*. Section 4.2.1 defines security types; Section 4.2.2 develops an algorithm for detecting ANI in the

---

\(^{1}\) Strictly speaking, we should say that, even with Turing-equivalent languages, most domains cannot be represented since they are inherently uncomputable. (Un)Fortunately, this is beyond the realm of computer scientists.

\(^{2}\) This paper does not rely on Abstract Interpretation; however, as pointed out in \cite{53}, Relaxed Non-Interference can be seen as a special case of ANI
FUN language (Section 2.1.2). Section 4.2.3 extends the discussion to a small subset of Java bytecode.

### 4.2.1 Representing abstract Information Flow with types

Dealing with Information Flow security properties involves comparing program executions. In facts, Non-Interference does not require that some program data be included in some range of values (which is, basically, the scope of standard type analysis). On the contrary, constraints are placed with respect to different program runs. For example, the definition of Non-Interference (Section 3.2) relates two executions which share the same input public values. Output public values are required to be the same; yet, there is no restriction on which values (in an absolute sense) they can take.

#### Types as sets of pairs

If we want to describe such a requirement by means of types, the most natural choice is to assign a set of pairs (instead of single values) to program data. We call such a set a security type. A pair identifies two values which can be possessed by a piece of data in any two program executions. Usually, we are interested in computing output values.

**Example.** If a variable \( x \) has security type \( \tau = \{ (n, n) \mid n \in \mathbb{Z} \} \), then it is guaranteed that, after any two executions, \( x \) has the same value:

\[
\forall v_1, v_2. \ (\llbracket P \rrbracket (v_1), \llbracket P \rrbracket (v_2)) \in \tau
\]

which is the same as \( \llbracket P \rrbracket (v_1) = \llbracket P \rrbracket (v_2) \) for any input values \( v_1 \) and \( v_2 \). This does not make assumptions on the (absolute) value of \( x \).

Let \( c \) be some program component containing relevant data (a variable, a lambda-expression, etc.). If \( c \) is public, Information Flow analysis should prove that, at the end of the program, the value of \( c \) cannot be distinguished if it could not be distinguished at the beginning. This amounts to say that, if the input security type for \( \tau(c)_{in} \) is a subset of \( \{ (v, v) \}_{V} \) (where \( V \) is some set of values), then, also, the output type \( \tau(c)_{out} \) only contains symmetric pairs. A security type which only contains pairs of the form \( (v, v) \) is said to be public; a program satisfies the Non-Interference property if

\[
(\forall c \text{ public}. \ \tau(c)_{in} \text{ is public}) \implies (\forall c \text{ public}. \ \tau(c)_{out} \text{ is public})
\]

The aim of this security type analysis is to infer security types for public and private program data and verify the above condition. Obviously, this is a theoretical model since such a set representation cannot deal with infinite sets of values. Its main purpose is to make clearer the representation of abstract properties via security types.
Since the definition of Non-Interference does not make assumptions on the value of private data, then any private component $d$ is given a non-public input security type

$$\tau(d)_{in} = V \times V = \{(v_1, v_2) \mid v_1, v_2 \in V\}$$

meaning that nothing is known of the compared value of $c$ in two arbitrary executions. No constraints are set on $\tau(d)_{out}$: it can by anything. Non-Interference holds if no pair $(v_1, v_2)$, with $v_1 \neq v_2$, flows to public components during the program execution.

**Abstract security types**

Let the abstract domain $A$ be generated by the upper closure operator $\rho$. If $\rho$ represents the observational power of an attacker $A$, then $A$ is not able to distinguish two values $v_1$ and $v_2$ if $\rho(\{v_1\}) = \rho(\{v_2\})$. In terms of security types, the value of a public $c$ cannot be distinguished in any two executions if its type is public, that is,

$$\tau(c) \subseteq \{\{(s, s) \mid s \in \{\rho(\{v\}) \mid v \in V\}\}$$

**Example.** Let $\rho_p$ denote the abstract domain of parity. Then, $\rho_p(c)$ is public if it does not contain the pairs ([even], [odd]) or ([odd], [even]), so that an attacker cannot distinguish two executions unless it can see something more than parity. In such a framework, a constant is assigned the type {([even], [even])} or {([odd], [odd])}, depending on its parity.

If the image of $\rho$ is a finite set of abstract values, then security types are finitely representable as sets of pairs of abstract values. Private components are given an input type $\rho(V) \times \rho(V)$, where $\rho(V) = \{\rho(\{v\}) \mid v \in V\}$.

**4.2.2 Higher-Order security types and type inference**

The most natural choice in defining Non-Interference for the FUN language [73] is to consider, as the relevant input program components, the environment variables, divided into public and private. The only output value is the result of evaluating a program (lambda-expression) $P$ in some environment. Non-Interference is satisfied by $P$ if attackers cannot distinguish between the value of an expression, computed in two different environments.

**Indistinguishability**

Indistinguishability of the (public) input is formalized by the predicate $\text{ENV}_E(\varepsilon_1, \varepsilon_2)$, where $\varepsilon_i$ are two environments mapping variables to (concrete values), and $E$ is an abstract environment, mapping variables to security types.
(depending on the functional type of variables and on whether a variable is
public or private). Two values \( v_1 \) and \( v_2 \) are indistinguishable with respect to
the type \( \sigma \) (written \( \text{ST}_{\sigma} (v_1, v_2) \)) if both \( v_1 \) and \( v_2 \) are represented by
\[
\text{ST}_{\sigma} (v_1, v_2) \equiv \exists (s_1, s_2) \in \sigma. (v_1 \in s_1 \land v_2 \in s_2)
\]
where
- abstract values \( s \) belong to the set \( \bar{\rho}(V) \) for some abstract domain \( \rho \) which
  models the precision of attackers;
- \( v \in s \) if the concrete value \( v \) is represented by the abstract value \( s \) (formally,
  \( v \in \gamma(s) \), see Section 3.1.2);
- \( (s_1, s_2)(\sigma) \) is obtained as the set \( \{ (s_1(t_1), s_2(t_2)) \mid (t_1, t_2) \in \sigma \} \) (the expres-
  sion \( s(t) \) is the abstract functional application of the abstract higher-order
  value \( s \) to the abstract value \( t \)).

The indistinguishability relation on environments extends the \( \text{ST} \) predicate:
\[
\text{ENV}_E (\epsilon_1, \epsilon_2) \equiv \forall x. \text{ST}_{E(x)} (\epsilon_1(x), \epsilon_2(x)).
\]
The Abstract Non-Interference condition on \( \text{FUN} \) is defined as follows.

**Definition 4.1 (Higher-Order Abstract Non-Interference).** Let \( P \) be a
functional expression, which is to be evaluated in an abstract environment \( E \).
Let \( \rho \) be the observational power of attackers on the output. Then, \( P \) satisfies
Abstract Non-Interference for \( E \) and \( \rho \) if
\[
\text{ANI}_{E, \rho} (P) \equiv \forall \epsilon_1, \epsilon_2. \text{ENV}_E (\epsilon_1, \epsilon_2) \Rightarrow \rho([P]_{\epsilon_1}) = \rho([P]_{\epsilon_2})
\]
If an attacker cannot distinguish two input environments \((\text{ENV}_E (\epsilon_1, \epsilon_2))\), nei-
ther he or she can distinguish the output computed value.

The type system

A (monomorphic) type system is provided, which infers a security type for \( P \).
The judgement \( E \vdash P : \{ \tau \} \), specifies that, in \( E \), the expression \( P \) is given
a (standard, monomorphic) type \( \tau \), together with a security type \( \sigma \). Some
typing rules are shown below.

\[
\begin{align*}
E \vdash n : \{\text{INT}\}_{\rho(n), \sigma(n)} & \quad \text{[const]} & E \vdash x : E(x) & \quad \text{[var]} \\
E \vdash b : \{\text{BOOL}\}_{\rho_b} & \quad E \vdash e' : \{\tau\}_{\rho'} & \quad E \vdash e'' : \{\tau\}_{\rho''} & \quad \text{[if]} \\
E \vdash \text{if } b \text{ then } e' \text{ else } e'' : \{\tau\}_{\rho_b \cup \rho'} & \quad \text{[if]} \\
E \vdash [F \leftarrow \sigma] : e : \sigma & \quad \sigma \text{ minimal} & \quad \text{[rec]}
\end{align*}
\]
The rule for constants assign a constant, public type to \( n \). Inferring the type for a conditional involves taking both branches into account. The recursion rule finds a minimal type among the fixpoints.

**Theorem 4.2.** Let security types and abstract domains be related by

\[
\rho \leadsto \sigma \equiv \forall v_1, v_2. \text{ST}_\sigma(v_1, v_2) \Rightarrow \rho(v_1) = \rho(v_2)
\]

so that \( \rho \) cannot distinguish values which belong to the same type \( \sigma \). Then, the following holds:

\[
\rho \leadsto \sigma \land E \vdash P : \{\tau\}_\sigma \implies \text{ANI}_{E,\rho}(P)
\]

The security type \( \sigma \) is an upper approximation of the possible values of \( P \), then the condition \( \rho \leadsto \sigma \) implies that the output cannot be distinguished by an attacker with precision \( \rho \). Note that, if, as it usually is, \( \sigma \) is built over \( \bar{\rho}(V) = \{\rho([v]) \mid v \in V\} \), then \( \rho \leadsto \sigma \) only if \( \sigma \) is public.

**An example**

The outlined type system is able to correctly deal with recursive, higher-order expressions like

\[
P \equiv \mu F. \lambda x. \text{if } x = 0 \text{ then } 2 \text{ else } 2 \ast y \ast F(x - 1)
\]

Here, \( y \) is a private variable; the observational power of attackers is \( \rho_y \).

The abstract domain is lifted the the functional order by means of domain operators. For example, the set of functions from int to int is divided into five abstract classes: functions (i) mapping all numbers to even numbers; (ii) leaving the parity unchanged; (iii) inverting the parity value; (iv) mapping all numbers to odds; (v) the remaining functions. On the concrete side, the value of \( P \) clearly depends on \( y \), then Non-Interference is not satisfied. On the other hand, let abstract properties be taken into account. For every value of \( y \), the recursive function \( F \) has an even result value, regardless of \( x \); then, it belongs to the abstract class (i). Consequently, ANI for the domain of parities is satisfied by \( P \).

**Discussion**

This type system can be practically improved in several ways. However, it is, in general, undecidable when the semantics domain is infinite. Difficulties are increased by the construction of higher-order domains, which are, generally, much more complex than the primitive domains. Some typing rules, such as the lambda-abstraction rule, use universal quantifiers on abstract values, thus possibly resulting in non-termination.
4.2.3 Stack-based security types and type inference

This approach can be also applied [74] to a small subset of Java bytecode, namely, the non-procedural, sequential part which is based on the use of an operand stack for evaluating expressions. A Control Flow Graph represents the control flow of programs (see Chapter 6 for a similar technique). A block is a sequence of non-jump instructions which are guaranteed to be executed sequentially (no exceptions). The security results obtained for a block are combined in compound subprograms.

Security types are defined as in higher-order ANI (clearly, without higher-order features). In addition, some auxiliary information (prevalues) helps to improve the type analysis by remembering the security conditions at the beginning of a block.

An abstract operand stack is used, which contains security types instead of concrete values. The abstract stack is manipulated according to the concrete one: when a binary operation is performed, two values are popped and the result (of combining their security types) is then pushed on the stack.

4.2.4 Brief discussion

The notion of security type which was presented in this section is interesting since it provides a first step towards an automatic representation of abstract security information. Next sections in this chapter will follow a different approach and deal with more complex languages. The focus will be on defining security properties whose analysis takes advantage of the language mechanisms. This could help to build an integrated analysis framework which checks Information Flow security properties together with standard correctness conditions (such as type safety) which are enforced by the usual design and execution frameworks.

4.3 A realistic approach to Information Flow

Most Information Flow properties were originally defined on simple languages, similar to IMP and FUN (introduced, respectively, in Section 2.1.1 and 2.1.2). In this case, programs work on amounts of data which are fixed throughout the execution; it is not possible to create or delete data by new variable declaration, explicit memory allocation/deallocation, object creation or garbage collection. Due to these simplifications, it is easy to split data into a public and a private part; the distinction between input and output values is also clear. Consequently, the foundational definitions of Non-Interference work well.

None of today real-life programming languages do match this framework. The ability to produce large and flexible programs is utterly increased by providing mechanisms which manipulate, at runtime, the amount of data which can be accessed and modified. These techniques result in a more complex
behavior of programs with respect to properties about the manipulation of program data. Information Flow is one of such properties; the analysis of realistic programming languages needs more comprehensive notions of Non-Interference to be defined. For example, the simple addition of dynamic variable declaration to IMP is a sufficient condition to make the original notion of Non-Interference somehow inadequate: the variable set is no longer constant during program execution.

In imperative languages, many features concur to increase the difficulties:

- the presence of procedures needs a consistent treatment of subprograms;
- variables can be declared everywhere in the code;
- program types are not limited to integer numbers;
- programs can receive and release information during their execution via I/O operations.

In addition, some problems are peculiar to the Object-Oriented framework:

- Classes identify new data types whose structure can be quite complex.
- Objects can be dynamically created, so that the amount of allocated memory is not fixed.
- Objects can be nested into other objects as field values.
- A class can inherit data or behavior from its superclasses.
- Polymorphism: when procedures take the form of instance methods, it is not statically decidable which method will be invoked at runtime.

The specification of the Java programming language [41] includes all these features; unused memory is periodically freed by a Garbage Collection operation. Java also includes interface declaration, exceptions, concurrency and many other mechanisms which will not be addressed here.

4.3.1 Which definition of Non-Interference

The definition of Non-Interference by Goguen and Meseguer (Section 3.2) is the guideline for a correct model of Information Flow analysis. However, due to its genericity and informality, the problem arises of tailoring the property to a specific programming language and execution environment. In particular, it must be made clear what observers can see of an execution framework; that is, the semantics which is used by observers in order to acquire knowledge about programs.

**Example.** Let \( \mathcal{P} \) be a program written in a variant of IMP with dynamic variable declaration. \( h \) and \( h1 \) are private variables.

```plaintext
if (h == 0) {
    h1 = 1;
} else {
    int i = 8;
    h1 = 2;
}
```
4.3 A realistic approach to Information Flow

This code fragment has the property of Non-Interference if the observer can only see public variables on input and output (input-output semantics), since i is no longer visible outside the else branch and there is not any harmful assignment depending on the value of h. However, two executions with different values for h can be distinguished by the observer if, for example, he or she can observe one of the following program properties:

- the value of variables at every program point (trace semantics), not only on input and output; in this case, the temporary value 8 for i would be detected;
- the amount of public allocated memory; in this case, the execution of the else branch results in one more defined memory location.

This discussion may seem useless if we limit ourselves to simple programming languages; however, it turns out to be crucial in an Object-Oriented framework, where dynamic memory allocation is a key feature.

Remark 4.3. The above example discusses Information Flow program properties with respect to which parts of the execution environment an observer is able to see. Orthogonally, the ANI weakened version of Non-Interference is parameterized on how an observer can see data, that is, the level of precision in observing public information. A working definition of Abstract Non-Interference in a real-life programming language should answer to both questions.

4.3.2 What is an NI-secure program

A Java program is a collection of inter-operating classes; each class has its own operations and contains its own data, either belonging to the class itself or to some of its instances (objects). The notion of Non-Interference we adopt here is quite close to the original one:

what an attacker can see at the end of the execution of a program does not allow him or her to acquire (abstract) information about secret input data

In order to adapt this generic notion to Java, we must make clear the following:

- what an attacker is and can do during the execution of observed programs;
- what the input and output of a program precisely are;
- which data an attacker can watch, in the sense of Example 4.3.1.

Actually, we will deal with the language introduced in Section 2.2, instead of the real Java programming language. Consequently, advanced Java features as exceptions, interfaces, inner classes and concurrency need not to be taken into account.
Programs and attackers

Usually, when modeling Information Flow analysis, the rôle of attackers is not explicitly specified: an attacker is simply an agent which can watch the execution of programs with the magnifying glass provided by some semantics. For example, in analyzing IMP programs, the semantics is often the input-output semantics on the values of non-protected variables. Then, attackers do nothing but watching the value of public data at some selected program points (the beginning and the end).

In our Object-Oriented framework, we choose to see attackers $A$ as programs running in the same environment as the program $P$ whose secrets are to be broken. $A$ is supposed to include classes and methods which can interact with $P$ classes in order to disclose their secrets. An attacker can be seen in two ways:

- $A$ can invoke static or instance methods of any class of $P$, and freely access to their non-protected static or instance fields. In this framework, $P$ can be seen as a class library whose functionalities can possibly release secrets. There is no need for $P$ to be a complete program with a main() entry procedure.
- $A$ is interested to the whole execution of programs (which are complete programs with main()); then, input and output program data are those which can be accessed respectively before and after executing main().

In both cases, Non-Interference holds if $A$ is not able, by accessing to the public output data of $P$, to acquire information about the private input data.

The input and output of programs

The picture outlined above relies on the existence of objects which encapsulate relevant program data before methods are actually executed. Existence of objects can be supposed when methods are called as in class libraries (see first formalization of attackers in the previous paragraph). On the other hand, in the case of a standalone Java executable file, the main() method has, as its only input, the string sequence args of its actual parameters.

Our choice is to suppose that main() begins by the fetchData() static method, which fetches all the useful input information via the invocation of some class methods or instance methods belonging to newly created objects. These methods can, for example, read files or perform I/O operations; the data which are fetched can be public or private. The part of the program which is relevant to Information Flow analysis begins after fetchData(); the input of a program is the amount of data which have been acquired by this method.

Symmetrically, the output of the program is written to persistent memory or external devices by means of the procedure releaseData(). The basic structure of main() is shown below.
static void main(String[] args) {

    // declarations of the variables which will
    // contain the fetched data

    fetchData();

    // the principal part of the program begins here

    ...

    // the principal part of the program ends here

    releaseData();
}

Example. Let $P$ be a program which begins its execution by reading the file in.txt, performs some computations and eventually writes the results into in.txt. The initial procedure fetchData() loads in.txt into an object in of some suitable class. Similarly, final data are released from an object out to out.txt by releaseData(). $P$ can generate dangerous information flows if out.txt can be read by an attacker, while in.txt cannot. In this case the security policy must consider in as private and out as public.

The rest of our program analysis will suppose the existence of relevant data before and after executing methods.

Program I/O

Direct Input/Output is one of the most usual features in realistic programming languages. It involves the possibility that data can be acquired or released at intermediate executions steps (not only in fetchData() or releaseData()).

Input/Output procedures are methods. We can suppose that:

- For input: data are initially acquired by fetchData() and made available to $P$ computations; this does not correctly describe program interactivity, since, in a real framework, program inputs may also depend on previous computations.
- For output: data which are output at runtime are stored into special objects which cannot be further modified, so that an illicit output can be detected by observing public data after execution.

Example. Let $P$ be the program

public C m() {
where `release()` is an Output procedure. The returned value of `m()` is clearly unaffected by the private value `h` (provided that `m1()` has no side effects); however, since the value of `h` is output before the termination, an attacker would be able to dangerously acquire protected information. We suppose that `release()` writes the result of `m1(h,l)` in a public object `o`; the Information Flow analysis will detect this forbidden flow in the final value of `o`.

**Public and private data**

The Java programming language comes with its own modifiers `public`, `private` and `protected`. These modifiers model security requirements: for example, a private field cannot be directly accessed from outside its class. The security property which is enforced by modifiers is a kind of access control [48]: direct access to data is limited to a certain set of agents (objects). However, information flow properties should not be limited to direct access to data: secret information cannot be propagated to the visible part of data. The following piece of code outlines the difference:

```java
public class D {
    public C l;
    private C h;
    public C m() { return h; }
}
```

This class declaration is perfectly legal; yet, the private field `h` is propagated via the return value of the (public) method `m()`.

For the purpose of information flow analysis, we add two further modifiers `H` and `L`, meaning, respectively, high-security and low-security level, which are orthogonal (although related) to private, public and protected. For example, the field declaration `private H C f` is legal.

Due to this choice, the definition of public and private data is related to class fields: if `H C f` is declared in `D`, then all the `C` objects stored in the field `f` of `D` objects are considered as private. The program memory is split into private and public locations. A location `l` is private if (i) its content is an object referred by some private field; or (ii) it is referred by a private or public field of some private location `l'`.

In the following, the notion of private and public data will refer to `H` and `L` instead of private and public Java modifiers.
The attacker semantics on observed programs

The previous paragraphs define the semantics of attackers in observing programs. Basically, it is an *input-output* semantics restricted to variables which are externally visible, are defined in `fetchData()` and released in `releaseData()` (then, also including explicit I/O procedures). Attackers cannot see:

- private data throughout the whole execution;
- temporary data which do not reach the end of program execution; they can be observed only if their content flows to public data which are eventually released in `releaseData()`;
- the memory state; for example, the amount of allocated data at a given program point;
- the behavior of programs with respect to computational complexity.

The last two items (which are exemplified in the following method definitions) represent some form of information leaking; we choose not to consider attackers which can exploit them.

This fragment describes how an attacker which can see the amount of allocated memory is able to acquire information about the private variable $h$: he or she can guess the value of $b(h)$ by observing whether a memory region has been allocated by `someMemoryAllocation()`.

```c
// example of Interference via memory allocation
void m() {
    if (b(h)) {
        someMemoryAllocation();
    }
}
```

If we can observe the execution time, we are able to guess whether the boolean guard was verified at this program point.

```c
// example of Interference via the observation // of computational complexity
void m() {
    if (b(h)) {
        someExponentialComputation();
    }
}
```

4.4 Class-Oriented Abstract Non-Interference

Section 4.3 dealt with the problem of specifying *what* attackers can see of the execution environment of programs. The outlined choices apply to any analysis...
Representing Abstract Non-Interference, in the sense of Section 3.2. Abstract Non-Interference is interested in how data can be observed by untrusted users; that is, the degree of precision attackers have in observing public information.

In some universe of entities, abstract properties identify sets of elements sharing some common behavior. In an Object-Oriented language, classes have a similar purpose: they identify collections of objects with a similar internal structure. Then, we model properties by means of classes. This allows to reduce the checking of properties to a type-directed program analysis. Classes are ordered by the subclass relation, that is, a class represents a superset of the objects represented by any of its subclasses. The subclass relation models sub-properties, that is, properties which are more precise since they are satisfied by a smaller set of elements.

Depending on the context, in this section the identifiers $C, D, \ldots$ will denote sets of semantic objects, class names or the abstract properties they model. The subclass relation of $C'$ with respect to $C$ is denoted by $C' \prec C$. Given a class $C$, the predicate $o : C$ holds if the object $o$ is in class $C$ (or, equivalently, if it satisfies the property $C$). $o :: C$ means that $o : C$ and there does not exist any $D \prec C$ such that $o : D$.

Remark 4.4. We use the $(\rho)P(\rho \rightsquigarrow \|\rho)$ version of Abstract Non-Interference, not excluding a priori that the observational power is not the same for all program.

4.4.1 Class hierarchies as abstract domains

In our Abstract Interpretation framework, given a set of objects of some universe $C$, the powerset $C = \wp(C)$ is the concrete domain. A class hierarchy with $C$ as its root identifies a subset of abstract properties $D \in C$, that is, an abstract domain $A$.

Example. Let $C$ be the set of integers. In Java, it is modeled by the class $C = \text{Integer}$. Let the semantic abstract properties of sign be modeled by $\rho_s$; it can be represented by the following class hierarchy:

$$\{\text{[Integer]}, \text{[Pos] } \prec \text{[Integer]}, \text{[Neg] } \prec \text{[Integer]}\}$$

In terms of program classes, the upper closure operator $\rho_A(C)$, for an abstract domain $A$ and an object set $C$, is defined as the smallest class $C_0 \in A$ such that every $c \in C$ satisfies $c : C_0$. The lub $\sqcup$ of a set of classes is the least superclass of all classes in the set. No class equivalent exists for $\cap$ (see Section 4.4.2). The set of program classes can be always seen as a domain whose top element is the class Object and the bottom element is an ad hoc empty class bot. We can suppose bot to exist, even if it cannot be declared in Java as a subclass of all classes (see below), since it plays no part in computations; therefore, it is not strictly required to exist.
4.4.2 Class intersection

Since abstract domains are closed under intersection (Section 3.1), for every classes $C_1$ and $C_2$ there must be a subclass representing $C' = C_1 \cap C_2$. Due to the property of class hierarchies, $C'$ should be a subclass of both $C_1$ and $C_2$. Unless $C_1$ be a subclass of $C_2$ or vice versa, $C'$ cannot be represented since the Java language has no mechanisms for multiple inheritance.

Consequently, representable abstract domains are those in which two classes (sets of concrete values) are either related by $\preceq$ or disjoint. If they are disjoint, $C_1 \cap C_2 = \emptyset$ can be represented by the bot class.

In the following, an abstract domain $A$ is **tree-like** if it satisfies $\forall a, b \in A. a \cap b \in \{a, b, \bot\}$, that is, if it is representable by a single inheritance class hierarchy. We choose this name since these domains are graphically similar to trees, provided $\bot$ is excluded from the representation. In domains which are not tree-like, some abstract values cannot be represented as classes.

**Example.** Let the class Integer have subclasses Even and Odd, representing parities, and Pos and Neg, representing signs. A domain $\rho_{pa}$ representing properties of numbers with respect to both sign and parity is not representable by extending the same Java class hierarchy with new classes. In facts, the class NegEven of negative even numbers should be a subclass of both Even and Neg.

In a language with multiple inheritance as C++, the domain of the above example would be representable (obviously, if attention is paid to the complicated language mechanisms).

4.4.3 Domains and program code

Representing properties as classes involves performing the program analysis with the help of a type-checking algorithm. Usually, a Java program comes with its own classes for organizing and storing structured data. In general, program analysis can be demanded for a property which is not represented by any program class.

**Example.** A program $\mathcal{P}$ contains the classes $E, E_1, E_2$ and $E_3$ with $E_1 \prec E$, $E_2 \prec E$ and $E_3 \prec E$; the abstract domain represented by this class hierarchy is the **program domain** $\mathcal{A}_{\mathcal{P}}$. Let $\mathcal{A}_{\hat{A}}$ be a domain which cannot distinguish between $E_1$ and $E_2$; on the other hand, let $\mathcal{A}_{\hat{A}}$ contain the properties $E_3$ and $E_5$, which are both more precise than $E_3$. If the program designer wants to analyze the program with respect to the **analysis domain** $\mathcal{A}_{\hat{A}}$, he or she must include the information of $\mathcal{A}_{\hat{A}}$ into the program classes. Then, he or she should build a new domain representing both $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\hat{A}}$. The graphic representation of domains is shown in Figure 4.1.

The program should be modified according to the new hierarchy:
Fig. 4.1. The modified hierarchy

- an object creation new \( E_3(...) \) should be replaced by new \( E_5(...) \) or new \( E_6(...) \) whenever one of the two sub-properties holds for the new object. It should be clear that the analysis designer must know, for some \( x \), whether \( P_{E_5}(x) \) and \( P_{E_6}(x) \) hold or not (i.e. whether \( x \) satisfies the abstract properties): it is part of his or her knowledge of the problem.
- The analyzer does not need to distinguish \( E_1 \) from \( E_2 \) since \( \mathcal{A}_P \) can only see \( E_4 \); then, any occurrence of \( E_1 \) or \( E_2 \) can be replaced by \( E_4 \) (for example, when inferring the type of expressions).
- overriding methods of inserted classes should incorporate the behavior of operators with respect to the properties to be checked. This point will be better explained in the example below.

Remark 4.5. In the above example, both \( \mathcal{A}_P \) and \( \mathcal{A}_A \) are downflat (see Section 4.4.2) and their union is also a domain; \( \mathcal{A}_P \cup \mathcal{A}_A \) is also downflat, since there is no \( p \in \mathcal{A}_P \) and \( a \in \mathcal{A}_A \) such that \( p \cap a \notin \{p, a, \perp\} \). Consequently, it is the reduced product of \( \mathcal{A}_P \) and \( \mathcal{A}_A \). In general, this could be false; in facts, it is reasonable (for simplicity) to require \( \mathcal{A}_A \) to be downflat (ANI uses, in practice, flat domains which are also downflat), but this condition does not rule out the possibility to have two incomparable, non-disjoint abstract values in \( \mathcal{A}_P \cup \mathcal{A}_A \).

4.4.4 Class extension

In the common practice of Object-Oriented code design, subclasses are not only devoted to identify subsets of objects. In general, the subclass objects have a different internal structure with respect to objects belonging to the
superclass. Modifying the structure of objects is done by adding or redefining class members, that is, fields and methods.

The meaning of redefined (overridden) members depends on the rôle of classes in the program.

- If a class $E' \prec E$ is part of the program classes $\mathcal{A}_P$, then its redefined members simply specify the behavior of $E'$ objects when it is different with respect to $E$. In this case, program analysis must take into account every method which can possibly be executed at runtime.

- If $E'$ belongs to $\mathcal{A}_h$ (that is, it is not part of the original program), then its members should incorporate the knowledge of the designer about the security policy to be enforced.

The second possibility is made clear by the example below: Example. Consider the class hierarchy of Example 4.4.1; by redefining methods, it is possible to specify the behavior of integer operations on subclasses. In particular, the Object-Oriented addition on integers can be specialized to non-negative numbers:

```java
class Integer {
    ...
    Integer add(Integer n)
    ...
}

class Pos extends Integer {
    ...
    Integer add(Integer n); // inherited
    Pos add(Pos n);
    Integer add(Neg n)
    ...
}
```

The redefined add() method encodes some knowledge of the behavior of addition on positive and negative numbers. For example, adding two non-negatives yields a non-negative; on the other hand, nothing can be said if a non-negative is added to a negative.

\[\square\]

### 4.4.5 Advantages and limitations

Modeling abstract properties via classes allows to reduce the process of computing and checking properties to well-known typing algorithms.

Theoretical definitions of Abstract Non-Interference and similar properties, which rely on performing operations on abstract domains, often use a set-theoretic representation of abstract values. For example, in the abstract domain of parity, there are abstract values identifying the infinite sets of even
and odd numbers. In principle, representing infinite sets is impossible. In- tensional, algorithmic definitions can deal with some simple cases (e.g. even numbers are those which yield no remainder when divided by 2), but in general the problem of verifying if an element satisfies a property is not decidable.

The modeling of abstract properties by means of classes relies on the set- theoretical flavour of Object-Oriented mechanisms. However, many Object- Oriented features do not match with this spirit. For example, the execution of some object method can result in a change in the behavior of the object itself with respect to some property.

Example. Let the class Mod3 model the properties contains an integer number which is divisible by 3.

```java
class Mod3 {
    int n; // this must be divisible by 3
    void m() { n++; }
}
```

It is clear that m() can break the property described by Mod3 because of the new value of n.

Moreover, it is not always possible to consider classes as subsets of their superclasses.

Example. The following classes (which use primitive types, not included in our language) show how it is possible to declare the subclass OddButLegal, whose instances are not a subset of Pos instances.

```java
class Pos {
    // positive integers
    protected int x;
    Pos() { x = 0; }
    void m() { x++; }
}
class OddButLegal extends Pos {
    // integers such that x <= -11
    OddButLegal() { x = -11; }
    void m() { x--; }
}
class SuperClass extends Pos {
    // integers such that x >= -5
    SuperClass() { x = -5; }
}
```

\[\Box\]
In facts, our properties are likely to be of the form contains a field $f$ of class $C$. Then, standard type checking would guarantee the property preservation. 

Example. Let $C_1$ and $C_2$ be subclasses of $C$. Then, the properties $D$, $D_1$ and $D_2$ are defined as contains $f$ of class $x$, where $x$ is, respectively, $C$, $C_1$ and $C_2$.

```java
class D { C f; C g; }
class D1 { C1 f; C g; }
class D2 { C2 f; C g; }
```

Subclasses $D_1$ and $D_2$ cannot contain methods which break their correspondent abstract property, since type integrity forbids an assignment to $f$ unless it is compatible with $C_1$ or $C_2$.

If we limit ourselves to properties of this form, the preservation of a property on a given object is automatically enforced by type checking mechanisms. Otherwise, if our purpose is to model data which can change their behavior with respect to properties, this can be done by created new objects of different class which acquire the invariant part of relevant data. 

Example. The following method perform a pseudo-modification of an object by replacing it with some new object with the desired property.

```java
D2 changeProperty(D1 d, C2 c) {
    D2 aux = new D2();
    aux.f = c;
    aux.g = d.g;
    return aux;
}
```

Clearly, changeProperty() cannot be a method of the class $D_1$, since an object is not able to modify its own class.

Under the condition of limiting abstract properties to constraints on the properties of object members, the subset relation is correctly formalized by subclasses: for example, the class $D_1$ in Example 4.4.5 identifies a subset of $D$, provided that the same condition is inductively satisfied by $C_1$ and $C$. 
Better be despised for too anxious apprehensions, than ruined by too
confident security.

Edmund Burke (1729 - 1797)

Misquotation is, in fact, the pride and privilege of the learned. A
widely-read man never quotes accurately, for the rather obvious rea-
tion that he has read too widely.

Hesketh Pearson, *Common Misquotations*

In this chapter, the main lines of an algorithm for checking Abstract Non-
Interference in JL programs are presented. This static analyzer computes a
boolean function representing the behavior of programs with respect to the
desired information flow properties. Boolean functions encapsulate informa-
tion about the propagation of secrets among program data. As in the previous
chapter, we intend typical numerical domains as they are defined in Section
3.1.7.

5.1 Preliminary definitions

5.1.1 Methods

Methods are declared as class members. The same method name can be as-
signed to several subprograms. In the following, no distinction will be made
between (i) different methods with the same name but different, incompat-
ible signatures; (ii) different instances of a method (obtained, for example,
by means of overriding). We simply consider the set $M_d$ as containing all
the subprograms which can be executed when invoking methods. An element
Program analysis for ANI

$d \in \mathcal{M}_d$ will be referred to as a method definition, a method instance (not to be confused with an instance method) or, simply, a method.

Example. Consider the following classes:

```java
class C {
    D m(...) { ... } // d_1
    E m(...) { ... } // d_2
    F n(...) { ... }
}
class C1 extends C {
    D m(...) { ... } // d_1'
    G p(...) { ... }
}
```

After these class declarations, $\mathcal{M}_d$ contains three method instances for the method name $m$. The method instance $d_1'$, which redefines the corresponding $C$ instance $d_1$, and $d_2$, which is another method tout court, are equally considered as distinct instances with respect to $d_1$.

In the following, our interest will be in computing which method instances can be executed at runtime when a method invocation occurs.

### 5.1.2 Type environments

Type environments $\varepsilon \in \mathcal{E}$ are functions from identifiers to types $C \in \mathcal{C}$. For every program point $p$ in a method body, a type environment is computed in order to approximate the possible runtime types that identifiers can have at $p$. The abstract semantics $\llbracket s \rrbracket_\varepsilon^s$ computes the type environment after the statement $s$, starting from $\varepsilon$.

Let the method instance $d$ be declared in the class $D$ as $C.m(P_1, \ldots, P_k)$, where $P_1 \ldots P_k$ are the expected types of parameters. When analyzing the body of $d$, the initial type environment $\varepsilon_0$ is as follows:

$$
\begin{align*}
\varepsilon_0(p_i) &= P_i & \text{parameters} \\
\varepsilon_0(v) &= \perp & \text{local variables} \\
\varepsilon_0(\text{this}) &= D \\
\varepsilon_0(u.f) &= \text{declared type for } f \text{ in } \varepsilon_0(u)
\end{align*}
$$

The abstract semantics computes an over-approximation of runtime types, in the form of type environments:
In the following, $T_{\varepsilon}(e)$ will often stand for $[e]_\varepsilon^{T}$. The following definition uses type environments in order to identify the set of method instances which can be executed at a given program point.

**Definition 5.1 (method instances).** Let the method invocation

\[ v.m(e_1, \ldots, e_k) \]

occur at the program point $p$, in the type environment $\varepsilon$. For every class $C \triangleright [v]_\varepsilon^{C}$, the set $\text{INST}_\varepsilon(C, m, e_1 \ldots e_k)$ contains the method instances for which the type of formal parameters is compatible with the type of $e_1, \ldots, e_k$ in $\varepsilon$. If $m$ is a static method, so that the invocation takes the form $C.m(e_1, \ldots, e_k)$, then $C$ is the only class to be considered. Intuitively, we say that the types $P_j$ of formal parameters are compatible with $\varepsilon$ if they are comparable (that is, $\preceq$ or $\succeq$) to the corresponding $\varepsilon(e_j)$, so that their runtime types $C_0 \preceq P_j$ and $C_0' \succeq \varepsilon(e_j)$ can be possibly the same.

### 5.1.3 Program identifiers

When executing a method, the program code of the method body can refer to a set of program identifiers. This set contains the formal parameters, the local variables and the special variable this. Moreover, for an identifier $v$ and a field name $f$, the field $v.f$ may be referred.

In Object-Oriented languages, due to the well-known aliasing problem, a modification in a variable can affect other identifiers which refer to the same object. We are interested in computing which program identifiers can be affected by an assignment. The program memory can be seen as a set $L$ of locations $l$, each belonging to a class $C_l(l)$. In a given type environment $\varepsilon$, a variable $v$ is assigned to a set of locations $\text{LOCS}_\varepsilon(v)$, namely, an upper approximation of its possible runtime locations. We define a Galois connection between sets of locations and types:
Definition 5.2.

\[
\begin{align*}
\triangledown(L) & \xrightarrow{\gamma} C \\
\alpha(L) & = \gamma \left( \{C_l(l) \mid l \in L\} \right) \\
\gamma(C) & = \{l \in \mathcal{L} \mid C_l(l) \subseteq C\}
\end{align*}
\]

As a soundness requirement, we have \(T_\epsilon(v) \supseteq \alpha(\text{LOC}_\epsilon(v))\), that is, a typing must be a correct description of runtime values.

Next, we compute the set of locations corresponding to identifiers and expressions in \(\epsilon\):

Definition 5.3. The set of locations corresponding to variables and expressions is computed as follows:

\[
\text{LOC}_\epsilon(l.f) = \{\text{the location referred by the } f \text{ field of } l\}
\]

\[
\text{LOC}_\epsilon(v) = \gamma(T_\epsilon(v))
\]

\[
\text{LOC}_\epsilon(e) = \gamma(T_\epsilon(e))
\]

\[
\text{LOC}_\epsilon(v.f) = \bigcup \{\text{LOC}_\epsilon(l.f) \mid l \in \text{LOC}_\epsilon(v)\}
\]

When assigning an identifier by means of the statement \(x=e\), data propagation to \(x\) may affect all the identifiers potentially sharing their location with \(x\). These identifiers are said to be the sharing identifiers for \(x\) and are computed by \(\text{SH}(x)\).

Definition 5.4 (Sharing identifiers). The set of sharing identifiers for an identifier \(x\) is defined as

\[
\text{SH}_\epsilon(x) = \{v\}
\]

\[
\text{SH}_\epsilon(v.f) = \{u.f \mid \text{LOC}_\epsilon(u) \cap \text{LOC}_\epsilon(v) \neq \emptyset\}
\]

where \(\epsilon\) is the type environment at the current program point. When a local variable \(v\) is assigned, the set of its locations is completely replaced by a new one, then no other identifiers need to be considered. On the other hand, when modifying the field \(f\) of \(v\), it is possible that the same field of some identifier \(u\) is also modified, because \(u\) and \(v\) can refer to the same object.

The definition of \(\text{SH}\) aims at dealing with aliasing. As it is presented in Definition 5.3, \(\text{LOC}\) is clearly a very poor choice for this purpose. In facts, \(x\) is supposed to share locations with \(y\) unless their types are completely unrelated.

\textbf{Example.} Let \(x\) and \(y\) be defined by the assignments \(x=\text{new } C(\ldots)\) and \(y=\text{new } C(\ldots)\). Clearly, \(x\) and \(y\) do not alias even if they have the same type.

The separation of types is too strong a requirement for dealing with aliasing; some more refined mechanism is needed. For example, the Hoare-like logic illustrated in Section 3.6.1 can be better in telling whether program identifiers are supposed to alias.
5.2 Abstract dependency calculus

The calculus of dependencies [14, 2] is an important task in program analysis. Data dependencies describe how the information propagates in the code of a program during its execution. A well-known application is the analysis of unused variables in compilers, which aims at simplifying the translated program by detecting and eliminating useless parts of the input program [3]. If program data are contained in variables, we say that the variable $x$ depends on $y$ at the program point $p$ if the value of $x$ at $p$ is affected by $y$.

Since data are mainly manipulated by variable assignments, we are particularly interested in deciding, in the statement $x := e$, which variables are relevant to the abstract value of the expression $e$. This is the base level in computing dependencies; subsequently, they need to be propagated in the program code by means of some control flow algorithm (which we suppose to exist and build a dependency graph, see below).

5.2.1 Preliminary notions

This section introduces some notions and notation conventions which will be useful throughout this section. We also describe analysis tools for improving the results obtained with the dependency calculus of Section 5.2.5.

Dependency graphs

A dependency graph can be built out of a program by means of control flow program analysis; it describes how data propagate during the execution. Following an approach similar to that of Program Slicing [72], we might be interested in computing dependencies on program statements: the statement $s_2$ depends on $s_1$ if some of the variables which are used inside $s_2$ (and not redefined before their use) are defined inside $s_1$, and this definition reaches $s_2$ in at least one possible execution path. Graph vertexes consist of program statements; an edge from $s_2$ to $s_1$ is added if there exists a dependency of $s_1$ on $s_2$.

Example. Consider the following code:

\begin{verbatim}
(0) : v := 1;
(1) : x := e_1;
(2) : y := e_2;
(3) : x := 0;
(4) : z := x + y;
\end{verbatim}

The statement (4) depends on (2) since the definition of $y$ is relevant to the expression $x + y$; similarly, (4) depends on (3). On the other hand, it does not depend on (0) because $v$ is not used in (4). Finally, there are no dependencies
of (4) on (1) since the definition of \( x \) does not reach the last statement (that is, there is only one path from (1) to (4), and another assignment of \( x \) occurs on this path). The dependency graph for this fragment would look as

\[
\begin{array}{c}
(0) \\
\downarrow \\
(1) \\
\downarrow \\
(2) \\
\downarrow \\
(3) \\
\downarrow \\
(4)
\end{array}
\]

There is a number of tools for building and analyzing dependency graphs; they allow to compute how variable definitions propagate to program statements. In the rest of this work, it is supposed that some standard technique is used for this purpose. The key point which characterizes our approach is the definition of what using a variable means. Therefore, we will focus on the in the small level: how to compute the set of used variables in the statement \( x := e \). As will be clear in the rest of this section, this set cannot be easily computed as \( \text{vars}(e) \) when abstract properties are considered.

### The evaluation of expressions

We refer to the IMP language of Section 2.1.1. For a program point \( p \) and a variable \( x \), \( [x]_p^\sigma \) is the set of values \( x \) can possibly take at \( p \), starting from the initial state \( \sigma \). Given an expression \( e \in \mathbb{E} \), we write \( e [x_1, \ldots, x_k] \) to make explicit the occurring variables (that is, \( \text{vars}(e) = \{x_1, \ldots, x_k\} \)). For \( e [x_1, \ldots, x_k] \), the functional application \( e^\sigma (v_1, \ldots, v_k) \) is the result of evaluating \( e \) when, for every \( i \), \( x_i \) is given the value \( v_i \) (in this case, variables are considered as an ordered sequence).

On the abstract side, when an abstract domain \( \mathcal{A} \) (with \( \rho \) denoting the upper closure operator on the concrete domain \( C \)) is given and no confusion arises about it, \( e^\mathcal{A} (V_1, \ldots, V_k) \) is the abstract evaluation of \( e \) on the abstract values \( V_i \) belonging to \( \mathcal{A} \). Note that, in the following, the evaluation \( \gamma^\mathcal{A} (\cdot) \) of a constant \( v \) will be often replaced by \( (\gamma^\mathcal{A} (v))^\mathcal{A} (\cdot) \) (for example, in the domain of signs, evaluating \( (6)^\mathcal{A} \) is equivalent to evaluating \( ([\text{pos}])^\mathcal{A} \)).

An abstract expression \( e^\mathcal{A} \) with \( k \) variables can be seen as a function mapping \( \mathcal{A}^k \) to \( \mathcal{A} \), defined as the best correct approximation (see Section 3.1.4) of \( e^\sigma \):

\[
e^\mathcal{A} (V_1, \ldots, V_k) = \alpha \left( \{ e^\sigma (v_1, \ldots, v_k) \mid \forall i. \ v_i \in \gamma(V_i) \} \right)
\]

\([x]_p^\sigma (\sigma)\) is the abstract counterpart of \([x]_p^\mathcal{A} (\sigma)\), that is,

\[
[x]_p^\sigma (\sigma) = \alpha \left( [x]_p^\mathcal{A} (\sigma) \right)
\]
5.2 Abstract dependency calculus

The abstract semantics

We define a sound abstract semantics to statically infer abstract properties of variables. Given a variable $x$ and a program point $p$, the abstract semantics, parameterized on the abstract domain $\mathcal{A}$, computes an over-approximation $V'$ of the minimal abstract value $V \in \mathcal{A}$ which safely represents all possible concrete values of $x$ at $p$, so that the following must hold:

$$V' \geq_{\mathcal{A}} \bigvee \{ \lfloor x \rfloor_0^\mathcal{A}(\sigma) \mid \sigma \text{ is an input state} \}$$

An abstract environment $\varepsilon \in \mathcal{E}$ is a function mapping variables to abstract values (see Section 5.1.2 for a similar notion of environments with types). It is computed compositionally: $S \llbracket s \rrbracket_\varepsilon$ builds the abstract environment at $p$, where $s$ is a statement, $\varepsilon_0$ is the environment before $s$ and $p$ is the program point after $s$. $e^\varepsilon(\sigma)$ and $e^\varepsilon(\varepsilon)$ are, respectively, shorthands for $e^\varepsilon(v_1, \ldots, v_k)$ and $e^\varepsilon(V_1, \ldots, V_k)$, where, for every $i$, $x_i$ has value $v_i$ in $\sigma$ and $V_i$ in $\varepsilon$. Abstract environments are computed in the standard way ($\varepsilon' \vee \varepsilon''$ is computed pointwise):

$$S \llbracket \text{skip} \rrbracket_\varepsilon = \varepsilon$$
$$S \llbracket x := e \rrbracket_\varepsilon = \varepsilon[x \leftarrow e^\varepsilon(\varepsilon)]$$
$$S \llbracket s_1 ; s_2 \rrbracket_\varepsilon = S \llbracket s_2 \rrbracket_{S \llbracket s_1 \rrbracket_\varepsilon}$$
$$S \llbracket \text{if } b \text{ then } s_1 \text{ else } s_2 \rrbracket_\varepsilon = S \llbracket s_1 \rrbracket_\varepsilon \lor S \llbracket s_2 \rrbracket_\varepsilon$$
$$S \llbracket \text{while } b \text{ do } s \rrbracket_\varepsilon = \text{LFP}_\epsilon \left( \lambda e'. \varepsilon \lor S \llbracket s \rrbracket_{e'} \right)$$

5.2.2 Abstract Dependencies

As explained before, we are interested in finding which variables affect the final value of $x$ in the assignment $x := e$. We write $x \in \text{REF}(e)$ if $x$ is relevant to $e$, consistently with the notation of Program Slicing. Most standard dependency calculi compute relevant variables for $e$ as $\text{vars}(e)$. This is clearly an approximation: for example, if $e$ takes the form $x + y - y$, its value does not depend on $y$, although the variable belongs to $\text{vars}(e)$. However, describing dependency by means of variable occurrence (that is, by only considering the syntactic level) is, in the average case, quite a good approximation of the real (semantic) notion of dependency, outlined in Definition 5.5.

**Definition 5.5 (concrete dependencies).** An expression $e$ is said to depend on $y$ (written $y \rightsquigarrow e$) if its value can change depending on the different values $y$ takes, that is, if there exist two values for $y$ which imply different values of $e$. 
Program analysis for ANI

\[ y \mapsto e[y, x_1 \ldots x_k] \]

\[ \iff \exists v', v'', \{u_1\} \ldots \{v_k\}. e^\rho(v', u_1 \ldots u_k) \neq e^\rho(v'', u_1 \ldots u_k) \]

The picture is quite different when abstract dependencies are considered. The abstract calculus of dependencies is a weaker version of its standard (concrete) counterpart: it is very often the case that \( e \) depends, with respect to the considered abstract property, on a strict subset of its variables. The following is a semantic definition of abstract dependencies:

**Definition 5.6 (abstract dependencies).** \( e \) abstractly depends on \( y \), with respect to an abstract property \( \rho \) (written \( y \triangleright e \)), if the property can change depending on the abstract value of \( y \).

\[ y \triangleright e[y, x_1 \ldots x_k] \iff \exists v', v'', u'_1 \ldots u'_k, u''_1 \ldots u''_k.
\alpha(u'_i) = \alpha(u''_i) = V_i \land e^\rho(\alpha(v'), V_1 \ldots V_k) \neq e^\rho(\alpha(v''), V_1 \ldots V_k) \]

If we compute abstract dependencies by only observing syntax, then a considerable loss of precision occurs. In facts, any purely syntactic approach looks at occurrence as the only criterion to decide relevance (formally, \( x \mapsto e \) iff \( x \in \text{VARS}(e) \)). As a consequence, the syntactic-directed result for any abstract property \( \rho \) would be the same as for concrete dependencies. Instead, we want abstraction to make a difference with respect to dependencies.

**Example.** Consider the expression \( e[x, y] \equiv (x \ast x + 1) \ast y \). The final value of \( e \) clearly depends on both \( x \) and \( y \), so that the syntax-directed dependency calculus yields a correct result (indeed, all occurring variables are relevant). On the other hand, abstract dependencies with respect to the sign domain \( s \) show how a syntactical approach may be insufficient. In facts, \( x \) has no effects on the sign of \( e \) (that is, on the abstract value of \( e^\rho \)); yet, we need semantics in order to detect such a false dependency.

The following is an easy property of abstract dependency:

**Proposition 5.7.** Let \( \rho' \) and \( \rho'' \) be two abstract domains such that \( \rho' \) is more precise than \( \rho'' \). Then, for every \( x \in \text{VARS}(e) \),

\[ x \triangleright^\rho e \Rightarrow x \triangleright^\rho' e \]

### 5.2.3 Computing an expression on its inputs

Given an expression, the aim of the abstract dependency calculus is to compute the set of variables which are indeed relevant to its abstract value. As
shown above, this is, in principle, a semantic task and cannot be done automatically for arbitrary expressions. In other words, our only possibility to exactly translate Definition 5.6 into an algorithm is to acquire information about the abstract evaluation of $e$ on every possible input.

**Example.** To compute the set of $\rho_s$-dependencies on an expression $e[x,y,z]$, we must compute $e^z$ on every possible sign of $x$, $y$ and $z$. Thus, the value of $e^z$ must be computed $3^3 = 27$ times. An occurring variable ($y$, in this example) is not relevant to $e^z$ if the result

$$e^z(V_x, [\text{neg}], V_z) = e^z(V_x, [0], V_z) = e^z(V_x, [\text{pos}], V_z)$$

holds for every $V_x$ and $V_z$, that is, changing the abstract value of $y$ does not affect $e^z$.

We assume to have an algorithm which computes $e^z$ on an abstract input. Intuitively, given an abstract domain $\rho$, $e^z$ models the best correct approximation of $e^z$, that is,

$$e^z(e) = \rho \left( \left\{ e^\sigma(\sigma) \mid \sigma \in \rho(e) \right\} \right)$$

For example, $e^z$ can have a set of abstract operators like $\oplus$ or $\otimes$, describing the behavior of $+$ or $\times$ in $\rho$:

- $[\text{pos}] \oplus [\text{pos}] = [\text{pos}]$
- $[\text{pos}] \otimes [\text{pos}] = [\text{pos}]$
- $[\text{neg}] \oplus [\text{neg}] = [\text{neg}]$
- $[\text{neg}] \otimes [\text{neg}] = [\text{pos}]$

Let $|e|^z$ be the computational complexity of $e^z$ on an expression $e$. In the example above, the abstract value of $e$ must be computed once for every abstract input. Consequently, this technique has exponential complexity $O \left( |e|^z \cdot \#(\mathcal{A})^k \right)$, where $k = \#(\text{vars}(e))$ is the number of variables occurring in $e$. Although, in common expressions, both $\#(\mathcal{A})$ and $k$ are usually quite small, an improvement is definitely needed. In facts, this calculus should also deal with procedures and functions, which possibly involve a larger set of variables.

An abstract value $V \in \mathcal{A}$ is called an *atom* if there is no other $V' \neq \bot$ such that $V' \leq V$. The predicate $\text{ATOM}(V)$ indicates atomicity.

**Assumption 5.1** In Definition 5.6, the focus is on the abstraction of singletons (see also Example 5.2.3, where $\top$ is not considered among the possible inputs). We assume that atomic values are those which can be obtained by the abstraction of singletons:

$$\text{ATOM}(V) \iff \exists v. V = \alpha \{v\}$$

1 We leave to Remark 5.8 the discussion on how $\geq$ instead of $=$ might hold in this formula.
The assumption holds if and only if atoms form a partition of the set of concrete values. This is not true for arbitrary domains. However, every domain \( \mathcal{A} \) can be transformed into another domain \( \mathcal{A}' \) which satisfies the property. \( \mathcal{A}' \) is obtained by adding to \( \mathcal{A} \), for every \( V' \in \mathcal{A} \) and \( V'' \leq V' \), the abstract value representing \( V' \setminus V'' \), that is, by adding complements. The domain \( \mathcal{A}' \) is said to be partitioning.

*Example.* Let the domain \( \mathcal{A} \) on natural numbers consist of the abstract values

\[
\begin{align*}
[2] &\equiv \{ n \mid n \mod 2 = 0 \} \\
[3] &\equiv \{ n \mid n \mod 3 = 0 \} \\
\end{align*}
\]

The only atom, \([6]\), does not represent all numbers. In order to satisfy the property, we must add

\[
\begin{align*}
[\neg(2, 3)] &\equiv \mathcal{Z} \setminus ([2] \cup [3])
\end{align*}
\]

In the new domain, \( \mathcal{Z} \) is indeed partitioned by the atomic abstract values \([6]\), \([3 \setminus 2]\), \([2 \setminus 3]\) and \([\neg(2, 3)]\).

If two concrete values are both described by the same atom, then they cannot be distinguished by any property of \( \mathcal{A} \). In computing \( e^2 \), we are interested in atomic inputs, that is, inputs \( \varepsilon \) which only consist of atomic values (\( \forall x. \text{ATOM}(\varepsilon(x)) \)). These inputs are those which can be obtained by abstracting a single concrete input (\( \forall x. \varepsilon(x) = \rho(\sigma(x)) \) for some \( \sigma \)).

### 5.2.4 Increasing efficiency

Instead of computing \( e^2 \) on every input, we aim to be more efficient. Several improvements can be introduced:

**Limiting the set of inputs**

Let the expression \( e[x,y,z] \) in Example 5.2.3 be computed at the program point \( p \). Suppose some static analysis technique (for example, an abstract semantics which over-approximates the runtime value of variables) can tell that the value of \( y \) at \( p \) is always positive. Then, the inputs for \( e^2 \) at \( p \) are limited to \( \{(V_x, \text{pos}, V_z) \mid V_x, V_z \in \mathcal{A}\} \), which contains \( 3^2 = 9 \) elements instead of 27. This additional knowledge can often considerably reduce the amount of computation to be performed.
5.2 Abstract dependency calculus

Computing $e$ on non-atomic inputs

Let $\rho_{\varepsilon}$ capture both the parity and the sign of numbers. The set of atomic values consists of $\text{poseven}$, $\text{negeven}$, $\text{posodd}$, $\text{negodd}$; atoms form a partition of numbers into four subsets. Given $E = \{\text{poseven}, \text{negeven}\}$ and $O = \{\text{posodd}, \text{negodd}\}$, the assertion

$$\forall V' \in E. \forall V'' \in O. \ e^\sharp (V_x, V', V'') = U$$

is implied by the more generic assertion

$$e^\sharp (V_x, \text{even}, \text{odd}) = U$$

since $\text{poseven}$ and $\text{negeven}$ (resp. $\text{posodd}$ and $\text{negodd}$) form a partition of $\text{even}$ (resp. $\text{odd}$). Consequently, if we can prove a result for some input $\varepsilon$, we do not need to prove more specific formulæ on more restricted inputs. Then, our purpose will be to infer properties on general inputs and consider sub-inputs only if necessary.

Remark 5.8. Actually, due to the incompleteness (in the sense of Abstract Interpretation) of most analyzers, it can happen that the above implication does not hold. For example, let the input $\varepsilon$ be partitioned by $\{\varepsilon_1, \ldots, \varepsilon_j\}$. Then, it is possible that the computed value for every $\varepsilon_i$ is $e^\sharp (\varepsilon_i) = U$, but $e^\sharp (\varepsilon) = U'$ with $U < U'$. However, in the following we will search for generic assertions in which $U_0$ is atomic; then, in this case, $U < U'$ cannot be true.

Example. Let $e[x] = x \times x + 1$ be evaluated in $\rho_{\varepsilon}^z$. Let $\varepsilon_p$ satisfy $\varepsilon_p(x) = \top$. $\varepsilon_p$ is partitioned by its sub-inputs $\varepsilon_1$, $\varepsilon_2$ and $\varepsilon_3$, such that $\varepsilon_1(x) = \text{neg}$, $\varepsilon_2(x) = \text{0}$ and $\varepsilon_3(x) = \text{pos}$. Suppose $e^\sharp$ uses the usual rules about product and sum:

$$\begin{align*}
\text{pos} \times \text{pos} &= \text{pos} & \text{neg} \times \text{neg} &= \text{pos} & \text{0} \times \text{0} &= \text{0} \\
\text{pos} + \text{pos} &= \text{pos} & \text{0} + \text{pos} &= \text{pos} & \text{top} \times \text{top} &= \text{top}
\end{align*}$$

This is enough in order to compute

$$e^\sharp (\varepsilon_1) = e^\sharp (\varepsilon_2) = e^\sharp (\varepsilon_3) = \text{pos}$$

since, for example,

$$e^\sharp (\varepsilon_1) = \text{neg} \times \text{neg} + \text{pos} = \text{pos} + \text{pos} = \text{pos}$$

Yet, the abstract evaluator would need some more knowledge in order to infer the general result $e^\sharp (\varepsilon) = \text{pos}$, since $\text{top} \times \text{top} + \text{pos} = \text{top}$. □
Analyzing $e$ compositionally

The syntactic structure of expressions can help in computing relevant variables. In particular, computations performed on subexpressions can turn to be useful in analyzing the main expression. For example, let $e[x_1, x_2, x_3, x_4]$ be the sum of $e'[x_1, x_2, x_3]$ and $e''[x_3, x_4]$. We have that $x_1 \in \text{REF}(e')$ is a necessary condition for having $x_1 \in \text{REF}(e)$. Similarly, $x_3 \in \text{REF}(e)$ only if $x_3 \in \text{REF}(e')$ or $x_3 \in \text{REF}(e'')$.

If it is possible, by analyzing $e'$, to assert $x_1 \notin \text{REF}(e')$, then $x_1$ does not need to be taken into account in the analysis of $e$.

Computational bounds

After formally introducing the algorithm, it will become clearer how the amount of computations can be reduced by placing bounds on the analysis of subexpressions. Take $e$ as defined in the previous paragraph: if the analysis of $e''$ turns out to be too expensive (that is, some fixed limit of analysis steps has been reached), then it can be stopped and $\text{VARS}(e'')$ can be returned. Obviously, this may result in a loss of precision, which, however, is sometimes worthy.

5.2.5 The algorithm

Notation

We write $e' \leq e''$, for any two inputs of $e'$, if $e'(x) \leq e''(x)$ holds for every $x \in \text{VARS}(e)$ (where $\leq$ is the partial order on the abstract domain). $\alpha(\sigma)$ is the pointwise extension of $\alpha$ to the variables of $\sigma$, and $\sigma \leq \varepsilon$ is a shorthand for $\alpha(\sigma) \leq \varepsilon$ or, equivalently, $\forall x. \alpha(\sigma(x)) \leq \varepsilon(x)$. When $e$ is clear by the context, $\forall x.$ is a shorthand for $x \in \text{VARS}(e)$. Finally, when the name of variables induces some ordering (as in $x_1, x_2$ etc.), $\varepsilon = (V_1, V_2, \ldots)$ means that $\varepsilon(x_i) = V_i$ for every $i$ (similarly for concrete inputs).

Computing the input set

Given a program point $p$, the abstract semantics of Section 5.2.1 computes an abstract environment $\varepsilon_p$. An environment can be seen as an input state for an expression, so that inputs and environments will be used interchangeably. When an expression $e$ is evaluated at $p$, the possible abstract inputs are safely approximated by $\varepsilon_p$: for every possible concrete input $\sigma$ at $p$,

$$\forall x. \alpha(\sigma(x)) \leq \varepsilon_p(x)$$

Then, the set of abstract inputs is $\text{INPUTS}_p(e) = \{ \varepsilon \mid \varepsilon \leq \varepsilon_p \}$. Atoms represent (abstract) constancy; that is, if we can infer an atomic value $V$ for $e$, then
every concrete value of $e$ is indistinguishable with respect to the abstract properties.

**Example.** Let $x$ and $y$ be two variables. If the static analysis computes an abstract value $V$ for both $x$ and $y$, then they are not guaranteed to have the same abstract property unless $V$ is an atom. For example, consider the properties of sign and parity, introduced in Section 5.2.4. Knowing that $x$ and $y$ are both even does not exclude that they can be distinguished by someone who can also observe signs.

Due to our representation of properties, this monotonicity result holds:

$$e' \leq e'' \implies \mathcal{A}^e(e') \subseteq \mathcal{A}^e(e'')$$

In facts, smaller abstract inputs represent smaller sets of concrete inputs, so that $f[e][\alpha]$ is a subset of $f[e'][\alpha]$ for $e \leq e'$. In the following, when we write $\forall \varepsilon$, it is implicit that $\varepsilon \leq \varepsilon_p$ for $\varepsilon_p$ computed at the current program point $p$.

For an abstract environment $\varepsilon$, $[\varepsilon]_x$ is the set of environments $\varepsilon'$ which are more restricted than $\varepsilon$ at $x$; that is, $\varepsilon'(x) < \varepsilon(x)$ and $\varepsilon'(y) = \varepsilon(y)$ for $y \neq x$. Let $X \in \phi(V)$; then, $[\varepsilon]_X$ is the straightforward set extension of the restriction operator.

**The calculus at work**

The proposed algorithm relies on the assumption introduced in Assumption 5.1: given an abstract value $V$, the set of abstract values which are less than or equal to $V$ is a covering (not necessarily a partition) of $V$, that is, all concrete values in $\gamma(V)$ are represented by at least one of its sub-values: $\gamma(V) = \bigcup \{ \gamma(V') \mid V' \leq V \}$. Consequently, the following holds for every $\varepsilon$:

$$\mathcal{A}^\varepsilon \left( \bigvee_{e' \leq \varepsilon} \{ \alpha \left( \mathcal{A}^\varepsilon \left( \varepsilon' \right) \right) \} \right) \Rightarrow \mathcal{A}^\varepsilon \left( \bigvee_{\sigma \leq \varepsilon} \{ \alpha \left( \mathcal{A}^\varepsilon \left( \varepsilon' \right) \right) \} \right)$$

Note that, given an algorithm for $\mathcal{A}^\varepsilon$, the analogous result on abstract computations does not hold, due to incompleteness (see Remark 5.8):

$$\mathcal{A}^\varepsilon \left( \bigvee \left\{ \mathcal{A}^\varepsilon(e') \mid e' \leq \varepsilon \right\} \right) \Rightarrow \mathcal{A}^\varepsilon \left( \mathcal{A}^\varepsilon(e) \right)$$

Anyway, since, by soundness, $\bigvee_{e' \leq \varepsilon} \mathcal{A}^\varepsilon(e') \leq \mathcal{A}^\varepsilon(e)$, we need to check the atomicity of $\mathcal{A}^\varepsilon(e')$ only if $\text{ATOM} \left( \mathcal{A}^\varepsilon(e) \right)$ cannot be directly proven. Conversely, since $\gamma(V) = \bigcup \{ \gamma(V') \mid V' \leq V \}$, the assertion $\bigvee_{e' \leq \varepsilon} \mathcal{A}^\varepsilon(e') < \mathcal{A}^\varepsilon(e)$ is a sufficient condition for taking $\bigvee_{e' \leq \varepsilon} \mathcal{A}^\varepsilon(e')$ as the computed value on $\varepsilon$ (thus avoiding incompleteness). This is properly the task of the function $\lfloor \cdot \rfloor^\varepsilon (\cdot) \downarrow$.

In Figure 5.1, we give a set of definitions for an approximated calculus of abstract dependencies.
Fig. 5.1. The calculus

\[\begin{align*}
\mathcal{E}^\mathcal{E} (\lambda) & : (\mathcal{E} \times \mathcal{E}) \rightarrow \mathcal{A} \\
\mathcal{E}^\mathcal{E} (\varepsilon \downarrow) & = \Lambda_{x \in \text{vars}(\varepsilon)} \left( \forall x' \in [\varepsilon], \mathcal{E}^\mathcal{E} (\varepsilon') \downarrow \right) \\
\text{ATOM} & : (\mathcal{E} \times \mathcal{E}) \rightarrow (\mathcal{A} \times \{\text{true}, \text{false}\}) \\
\text{ATOM} (e, \varepsilon) & = (\mathcal{E}^\mathcal{E} (\varepsilon \downarrow), \text{ATOM} (\mathcal{E}^\mathcal{E} (\varepsilon \downarrow))) \\
\text{REF3} & : (\mathcal{E} \times \mathcal{E} \times \mathcal{F}(\mathcal{V}) \times \mathcal{F}(\mathcal{V})) \rightarrow (\mathcal{E} \times \mathcal{F}(\mathcal{V})) \\
\text{REF3} (e, \varepsilon, X, Y) & = \text{match} \text{ATOM} (e, \varepsilon) \text{ with} \\
& (V; \text{true}) \rightarrow (V, X) \\
& (V; \text{false}) \rightarrow \bigcap_{y \in Y} \left( \bigcup_{x' \in [\varepsilon], x} \text{REF3} (e, \varepsilon', \{y\} \cup X, Y) \right) \\
\text{REF2} & : (\mathcal{E} \times \mathcal{E} \times \mathcal{F}(\mathcal{V})) \rightarrow (\mathcal{E} \times \mathcal{F}(\mathcal{V})) \\
\text{REF2} (V, \varepsilon, Y) & = \text{REF3} (V, \varepsilon, \emptyset, \emptyset) \\
\text{REF2} (z, \varepsilon, Y) & = \text{REF3} (z, \varepsilon, \emptyset, \{z\}) \\
\text{REF2} (e_1 \odot e_2, \varepsilon, Y) & = \text{let} (e'_1, Z_1) = \text{REF2} (e_1, \varepsilon, Y) \text{ in} \\
& \text{REF3} (e'_1 \odot e'_2, \varepsilon, \emptyset, Z_1 \cup Z_2) \\
\text{REF} & : \mathcal{E} \rightarrow (\mathcal{E} \times \mathcal{F}(\mathcal{V})) \\
\text{REF} (e) & = \text{REF2} (e, \varepsilon_p, \text{vars}(e))
\end{align*}\]

**Code analysis**

Definitions are translated (excluding optimizations) from a working implementation (Appendix A), written in the Haskell [62] functional language.

The function \text{ATOM} (e, \varepsilon) checks if it is possible to compute an atomic abstract value for e on \varepsilon, and computes it. As said above, the role of \mathcal{E}^\mathcal{E} (\varepsilon \downarrow) is to produce, when possible, an assertion on the larger input, starting from those on its sub-inputs. If \text{ATOM} (e, \varepsilon) is \( (V; \text{true}) \), then V is atomic; this means that \varepsilon is precise enough to infer an atomic value for e. Consequently, e only depends on the variables which are specialized in \varepsilon, with respect to \varepsilon_p.

**Example.** Let \varepsilon_p be the most general input, and \varepsilon \in [\varepsilon_p]_{x,y} be specialized on x and y, that is, \varepsilon(x) = V_x, \varepsilon(y) = V_y and \varepsilon = \varepsilon_p elsewhere. If \mathcal{E}^\mathcal{E} (\varepsilon) = V_e is atomic, then there is no possibility that any \( z \notin \{x, y\} \) is relevant to e, since, by the monotonicity of \mathcal{E}^\mathcal{E},

\[
\forall V_z \leq \varepsilon_p(z). \mathcal{E}^\mathcal{E} (\varepsilon [z \leftarrow V_z]) = V_e
\]

This means that different values of z cannot make a difference in the observed value of e. \(\square\)
Remark 5.9. Expressions are evaluated starting from $\varepsilon_p$, then some sub-inputs of $\lambda x. \top$ are not considered (they should be if $e$ would be considered as a standalone entity). Actually, this choice is correct since, in program analysis, we are interested in expressions as parts of programs. Consequently, it makes sense to ignore inputs which can never occur in concrete executions.

The function $\text{ref3}$ takes an expression, an environment and two sets of variables $X$ and $Y$. These sets represent, respectively, a temporary lower and upper approximation of the relevant variables of $e$. For example, the call of $\text{ref3}$ in the pattern $\text{ref2}(z, \varepsilon, Y)$ takes $Y = \{z\}$ since $z$ is the only variable which can be relevant to the expression $z$. Moreover, in the pattern $\text{ref2}(e_1 \otimes e_2, \varepsilon, Y)$, $Z_1 \cup Z_2$ is taken by $\text{ref3}$ since the relevant variables of $e_1 \otimes e_2$ are less than or equal to those of $e_1$ or $e_2$. The use of $Y$ only improves efficiency; $\text{vars}(e)$ could be taken instead. On the other hand, in $\text{ref3}(e, \varepsilon, X, Y)$, the set $X$ consists of the variables which are more specialized in $\varepsilon$ than in $\varepsilon_p$. Initially, $X = \emptyset$. If $X$ is enough to infer a ground value for $e^\sharp(\varepsilon \downarrow)$, then it is returned. Otherwise, a variable $y$ is added to $X$ and recursion is invoked on the restrictions of $\varepsilon$ on $y$.

In $\text{ref2}$, recursion is performed on the structure of expressions; conversely, $\text{ref3}$ recurs on the sub-inputs which are obtained by specializing the parameter on each $y \in Y$. The motivation for taking the intersection on $y$ of these sets is made clear by the Corollary 5.12 below.

Definition 5.10. Abstract dependency can be extended to sets $X \subseteq \text{vars}(e)$.

\[ X \not\rightarrow e^\sharp [X, y_1, \ldots, y_k] \iff \exists \{v'_x, v''_x\}_{x \in X, U_1 \ldots U_k} \text{ ground} . \]

where $\{\cdot\}_{x \in X}$ is to be seen as a tuple. Clearly, $[x] \not\rightarrow e$ iff $x \not\rightarrow e$ (Def. 5.6). Given a tuple $X$ of variables and a tuple $V$ of abstract values, $\varepsilon[X \leftarrow V]$ is the set extension of the environment updating: each $x \in X$ is updated with the corresponding $V \in V$. Two inputs $\varepsilon'$ and $\varepsilon''$ agree on $Z$ (written $\varepsilon' =_Z \varepsilon''$) if, for every $z \in Z$, $\varepsilon'(z) = \varepsilon''(z)$ holds. $=_Z$ is a shorthand for $=_{\text{vars}(e) \setminus Z}$.

Theorem 5.11.

\[ \neg (X \not\rightarrow e) \wedge \neg (Y \not\rightarrow e) \implies \neg (X \cup Y \not\rightarrow e) \]

(Otherwise stated: $X \cup Y \not\rightarrow e \implies X \not\rightarrow e \lor Y \not\rightarrow e$)

Proof. For each $e$, let $E^e_X$ be the set $\{\varepsilon \mid X \leftarrow \mathcal{V} \mid \mathcal{V} \in \mathcal{A}(X)\}$. We have

\[ \neg (Z \not\rightarrow e) \iff \forall \varepsilon. \forall \varepsilon', \varepsilon'' \in E^e_Z. e^\sharp(\varepsilon') = e^\sharp(\varepsilon'') \]

Then, the right hand sides of the formulae hold by hypothesis for both $X$ and $Y$. The elements of $E^e_Z$ are exactly those which agree with $e$ on $\text{vars}(e) \setminus Z$. 

5.2 Abstract dependency calculus 79
We must prove that \( \forall \varepsilon_1, \varepsilon_2 \in E^0_{X \cup Y}. \varepsilon^1 (\varepsilon'_1) = e^2 (\varepsilon_2) \). We already know that \( \varepsilon_1 = \varepsilon (X \cup Y) \varepsilon_1 \) and \( \varepsilon' = \varepsilon (X \cup Y) \varepsilon_2 \). Moreover, \( \varepsilon' \) can be chosen to verify \( \varepsilon'_1 = \varepsilon(X \setminus Y) \varepsilon_1 \) and \( \varepsilon'_2 = \varepsilon(Y \setminus X) \varepsilon_2 \). Hence, both assertions \( \varepsilon'_1 = \varepsilon(X \setminus Y) \varepsilon_1 \) and \( \varepsilon'_2 = \varepsilon(Y \setminus X) \varepsilon_2 \) hold. By hypothesis, \( \{\varepsilon'_1, \varepsilon_1\} \subseteq E^0_Y \), so that \( e^2 (\varepsilon'_1) = e^2 (\varepsilon_1) \). Similarly, \( e^2 (\varepsilon'_2) = e^2 (\varepsilon_2) \) and, by transitivity, \( e^2 (\varepsilon_1) = e^2 (\varepsilon_2) \).

**Corollary 5.12.** Given \( e, \varepsilon, X \) and \( Y \), there is a minimal element (with respect to set intersection on variable sets) for the set of sets

\[
\left\{ \bigcup_{\varepsilon' \in |e|_p} \text{ref3} (e, \varepsilon', X, Y) \right\}_{\varepsilon \in Y}
\]

Therefore, taking the intersection on \( y \) in the definition \( \text{ref3} \) is meaningful.

**Substitution of subexpressions**

This algorithm has an important feature: it computes, together with the set of relevant variables, a modified expression which is equivalent to the given one, with respect to the abstract domain, but possibly simpler. This is done by replacing constant (that is, without relevant variables) subexpressions with the corresponding atomic abstract values (see the first pattern in the definition of \( \text{ref3} \)). The following property holds:

**Proposition 5.13.** Let \( \varepsilon' \) and \( X \) be the expression and the set of variables obtained by computing \( \text{ref} (e) \). Then, \( y \notin \text{vars} (\varepsilon') \) implies \( y \notin X \).

**Proof.** It follows easily by observing that, if \( y \) does not occur in \( \varepsilon' \), then it only occurs in constant (without relevant variables) subexpressions of \( e \).

Note that the other direction of the implication is not true. As a counterexample, let \( e = \text{fst}(x_1, x_2) \), with \( \text{fst} = \lambda (x, y). x \); in this case, \( x_2 \) is clearly not relevant to \( e \), but \( \varepsilon' = e \) since the second subexpression of \( e \) is not constant by itself.

**An example**

Let \( e [x_1, x_2, x_3] = (6 \cdot x_1) + (4 \cdot x_2 \cdot x_2) + x_3 \) be analyzed in \( \rho_p \). Suppose some information can be inferred on the initial input \( \varepsilon_p \), namely, that \( x_3 \) is always negative at \( p \). Therefore, \( \varepsilon_p = ([\top], [\top], \text{neg}) \). Clearly, \( e \) is not constant in \( p \) with respect to the abstract properties: \( e^2 (\varepsilon_p) = [\top] \) (no matter how much the algorithm for computing \( e^2 \) is refined), so that both parity and sign can change depending on the concrete value of variables. However, we can infer some properties for any of the addenda:
5.2 Abstract dependency calculus

- $6 \times x_1$ is always even: $(6 \times x_1)^\dagger (\varepsilon_p) = [\text{even}]$.
- $4 \times x_2 \times x_2$ is non-negative and even.
- $x_3$ is negative, due to the restriction on $\varepsilon_p$.

Every variable only occurs in one addendum, then it is relevant only if the addendum is. Here, we have that the information about $6 \times x_1$ and $x_3$ is not enough to mark the variables as irrelevant ($[\text{even}]$ and $[\text{neg}]$ are not atomic). On the other hand, $x_2$ cannot affect the abstract value of the second addendum, then it is not relevant. The algorithm is able to produce this result, since $e^\dagger ((V_1, [\top], V_2) [\top])$ is atomic for every atomic values $V_1$ and $V_2$. In facts, the result of $\text{Ref}(e)$ is $((6 \times x_1) + [\text{poseven}] + x_3, \{x_1, x_3\})$. Note that, in the (less precise) abstract domain of parity, $x_3$ is the only relevant variable, since $6 \times x_1$ is constant.

**Soundness**

The soundness result we want to obtain is that, if $x \overset{\rho}{\sim} e$, then $x$ belongs to the set of variables which is computed by $\text{Ref}(e)$ on the abstract domain $\rho$. Since the main part of the computation is implemented by $\text{Ref3}$, the following lemma has to be proven.

**Lemma 5.14 (soundness of $\text{Ref3}$).** Let $e$ be an expression and $Y$ be the set of variables which has been computed on its subexpressions (as $Z_1 \cup Z_2$ in the definition of $\text{Ref2}$). Then

$$x \overset{\rho}{\sim} e \implies x \in X \text{ where } (e', X) = \text{Ref3}(e, \varepsilon_p, \emptyset, Y)$$

**Proof.** By induction on the structure of expressions. If $e$ is a constant, then its abstract value is always ground and $\text{Ref3}$ correctly returns $\emptyset$. If $e = y$, then $y$ is relevant if and only if $\varepsilon_p(y)$ is not ground; again, $\text{Ref3}$ is sound.

Let $e$ be the composition of a set $\{e_i\}$ of subexpressions. By inductive hypothesis, we can take $Y$ as a sound over approximation of the relevant variables of $e$ (since $\text{Ref3}$ has been invoked on $e_i$). Suppose, by contradiction, that $x$ does not belong to the computed set. We define two predicates:

$$\pi(Z) \equiv \text{Ref3}(e, \varepsilon_Z, Z, Y) \text{ computes } Z$$

Where $\varepsilon_Z \in [\varepsilon_p]_Z$. $\pi(Z)$ is equivalent to say that $e^\dagger ([\varepsilon_Z] [\top])$ is atomic.

$$\mu(Z) \equiv (\pi(Z) \lor \exists z \neq x. \mu(z \cup Z)) \land x \notin Z$$

The predicate $\mu(\emptyset)$ means that some set $Z$, which does not contain $x$, is returned by the $\text{Ref3}$ algorithm (note that the existential quantifier corresponds to set intersection in the definition of $\text{Ref3}$). Consequently ($\text{snd} = \lambda(a, b). b$, $x \notin \text{snd}(\text{Ref3}(e, \varepsilon_p, \emptyset, Y))$ implies $\mu(\emptyset)$. The base of $\mu$ recursive definition is $\pi(Z) \land x \notin Z$ for some $Z$ (the computed set). It is easy to see that $\pi(Z)$ implies that any variable outside $Z$ is not relevant to $e$ (see Definition 5.10), then $x$ is not relevant. □
Theorem 5.15 (soundness). Let $\rho$ be the abstract domain used in computing $e^\rho$, and $x \in \text{vars}(e)$. The result of $\text{ref}$ is a sound approximation of the relevant variables in $e$, under the assumption that the inputs to $e$ are restricted to those which are represented by $\varepsilon_p$.

$$x \not\in_e e \implies x \in X \quad \text{where} \quad (e', X) = \text{ref}(e)$$

Proof. It follows quite easily from the definition of $\text{ref}$ and $\text{ref}^2$, and Lemma 5.14.

Complexity issues

In the worst case, this algorithm is exponential (see Section 5.2.3. However, its complexity can be much better if some information can be inferred on $\varepsilon_p$ and there are subexpressions which are equivalent to constants.

This algorithm can easily be equipped with a computational bound: if, after consuming $N$ time-resources, $\text{ref}^3(e, \varepsilon, X, Y)$ has not terminated its execution, its evaluation can be interrupted and the upper approximation $Y$ can be returned (and made available on the recursion stack for the rest of the computation).

5.2.6 Discussion

This section defines an algorithm for computing abstract dependencies in expressions. In order to improve its efficiency, it relies on the structure of the abstract domain in which properties are described. The algorithm computes an upper approximation of the set $X$ of relevant variables, together with a simplified expression $e'$, which is obtained by replacing constant subexpressions. Even if expressions usually contain a few variables, this method can be also applied to functional subroutines (which are, usually, larger than expressions). Moreover, side effects can be taken into account by keeping track of the variables which are used inside procedures. In view of its application to several Program Analysis techniques, the algorithm should be modified in order to have $\text{vars}(e') = X$ where $(e', X)$ is the result of $\text{ref}(e)$; further research has to be carried out in this direction.

In the following sections, we denote by $\text{ref}_{\varepsilon_p}(e)$ (or $\text{ref}(e)$, when the input is implicit) the result of the $\text{ref}(e)$ algorithm on the abstract input $\varepsilon_p$.

In view of the properties as classes correspondence, we can consider the abstract domain as a Java class hierarchy; this does not invalidate the results obtained in this section.
5.3 Boolean functions

In this section, a formalism to describe the information flow behavior of programs is presented. Boolean functions [28] keep track of the propagation of secret information in program data. A boolean function is a first-order logic formula ranging on a set of basic predicates: the boolean variables.

**Definition 5.16 (Boolean variables).** Boolean variables are identifiers with a subscript:

\[ B = \{ x_j \mid x \text{ is an identifier, } j \text{ is a subscript} \} \]

The set of subscripts contains \( i \) and \( o \), standing for input and output, together with fresh subscripts which are created when required by the rules for computing boolean functions. Subscripts correspond to program points: for example, when analyzing a statement \( s \), the subscript \( i \) refers to the program point immediately before \( s \) (its input state).

Boolean variables can take, as their values, TRUE or FALSE; \( x_j \) is true if data identified by \( x \) can have a secret informational content at the program point corresponding to \( j \). A boolean variable corresponding to the program input (\( x_i \) for some \( x \)) is true if and only if the identifier \( x \) belongs to the private part of program data (see Section 4.3.2).

**Example.** In analyzing the program \( P \), suppose \( x_i \land \neg y_i \) holds, so that \( x \) is private and \( y \) is public. Let \( t \) and \( u \) be, respectively, the subscripts corresponding to immediately before and immediately after a variable assignment \( s \equiv (y := x) \). Then, provided \( x_t \) holds (that is, \( x \) has not lost its secret content in any program point before \( s \)), \( y_u \) is true because the secrets of \( x \) flowed into \( y \). If no further modifications occur to \( y \), then \( y_o \) is true. Consequently, there is a forbidden information flow from \( x \) to \( y \), because a public identifier is not guaranteed to be secret-free in the output (this is described by the formula \( \neg y_i \land y_o \)).

An identifier \( x \) can contain secrets if some of the data represented by \( x \) belong to the private part of program data, or if they have been assigned some value depending on private data. We say can contain secrets, instead of contains secrets, because, actually, a dangerous assignment is not a sufficient condition for a forbidden flow to exist. Yet, the analysis is interested in the negation of this condition, that is, that data cannot contain secret information.

A typical form for boolean functions is \( x_o \leftrightarrow x_i \), where \( \leftrightarrow \) is logical equivalence. This means that the statement does not modify the information flow behavior of \( x \), that is, \( x \) can contain secrets after \( s \) if and only if it can contain secrets before.

**Definition 5.17 (boolean function substitutions).** Given a formula \( \psi \), the formula \( [\psi]_{y_o}^{x_i} \) is obtained by variable substitution, that is, by replacing in
ψ all occurrences of \( x_t \) with \( y_u \). The subscript substitution \([ψ]^t_u\) is obtained from ψ by replacing all the occurrences of the subscript \( t \) with \( u \), that is, all variables \( v_t \) with their corresponding \( v_u \).

\[
[ψ]^x_{y_u} = ψ[x_t \rightarrow y_u] \\
[ψ]^t_u = ψ[v_t \rightarrow v_u] \quad \text{for each } v
\]

The special variable \( \text{RET}_\alpha \) refers to the result of evaluating an expression.

### 5.3.1 Models and Abstract Non-Interference

A model for a boolean function \( ψ \) is an assignment \( M \) of boolean variables making \( ψ \) true. \( ψ \) is satisfiable if it has models.

**Example.** Consider the IMP assignment \( x := f(y,z) \). The corresponding boolean function is

\[
ψ = (x_o \leftrightarrow y_i) \land (y_o \leftrightarrow y_i) \land (z_o \leftrightarrow z_i)
\]

meaning that the output value of \( x \) can contain secrets if either \( y \) or \( z \) can in the input. The models for \( ψ \) are \( (v_s \in M \text{ if and only if } v_s \text{ is true in } M) \)

\[
[ψ] = \left\{ 0, \{y_i, x_o, y_o\}, \{z_i, x_o, z_o\}, \{y_i, z_i, x_o, y_o, z_o\} \right\}
\]

As an example, there are no models \( M \) such that \( \{y_i, z_i\} \cap M \neq \emptyset \) but \( x_o \notin M \): it is not possible that either \( y \) or \( z \) can contain secrets in the input if \( x \) cannot in the output.

Only a subset of the models has to be considered in Information Flow analysis: those which are consistent with the division of identifiers into public and private (described by the truth values of \( v_i \)).

**Definition 5.18.** Let \( Ψ(\mathcal{P}) \) be the boolean function associated with \( \mathcal{P} \). Non-Interference holds for \( \mathcal{P} \) if

\[
Ψ_0(\mathcal{P}) \equiv \left( \bigwedge_{x \in L} \neg x_i \right) \land \ Ψ(\mathcal{P}) \land \left( \bigvee_{x \in L} x_o \right)
\]

is not satisfiable. In other words, \( \mathcal{P} \) has the Non-Interference property if every model of its boolean function which is consistent with \( L \) and \( H \) (\( \bigwedge_{x \in L} \neg x_i \)) does not allow any dangerous flows to public program identifiers (\( \bigvee_{x \in L} x_o \)).

In the above example, \( ψ \) is unsatisfiable if \( x \in H \) or \( y, z \in L \), that is, if the assignment is not dangerous since \( x \) is a private identifier or the assigned expression does not contain secrets.
5.4 Source code analysis via boolean functions

Example 5.3.1 refers to an imperative language where variables always denote the same location through the whole execution. Things get more complicated when dealing with an Object-Oriented programming language: identifiers $v$ or $u.f$ can share the same location, so that the effects of an assignment can be relevant to more than one identifier. Moreover, the boolean function for the IMP assignment was computed assuming, as in standard Non-Interference, that both $y$ and $z$ are relevant to $x$. As discussed in Section 5.2, we need to take Abstract Dependencies into account in order to model ANI. The function $\Psi(\mathcal{P})$ computes the boolean function which represents the information flow properties of $\mathcal{P}$ with respect to some assumption on the observational power of attackers.

5.4.1 Analysis of statements

This section presents a compositional abstract semantics for computing the information flow behavior of statements. In the following, $\varepsilon$ is the type environment computed immediately before the statement to be analyzed. Definitions 5.1, 5.20 and 5.21 provide the necessary background to the rules for computing $\Psi$. They should be read whenever required by Definition 5.22.

**Definition 5.19 (Method instances).** Let $\Psi(d)$ be the boolean function computed for a method instance $d$. The function $\Psi''(C.m, e_1, \ldots, e_k)$ computes $\psi$ for a method invocation of $m$ with actual parameters $e_1, \ldots, e_k$.

$$\Psi''(C.m, e_1, \ldots, e_k) = \bigvee_{d \in \text{inst}(C.m, e_1, \ldots, e_k)} \Psi(d)$$

Since it is not statically known which $d$ will be executed, the resulting boolean function is the disjunction on all the possible instances.

Note that $\psi' \lor \psi''$ is unsatisfiable if both $\psi'$ and $\psi''$ are; therefore, disjunction over $d$ is a conservative, sound choice (see Definition 5.18: Non-Interference must hold for every execution).

**Definition 5.20 (Implicit flows).** When the control flow of a program cannot be statically decided, as in conditionals or loops, the identifiers $G$ in the boolean guard can determine which path will be taken at runtime. Then, if some $x \in G$ contains secrets, there is an implicit information flow from $G$ to identifiers which are defined in some of the branches, since information about the values in $G$ can be acquired by watching which path is executed. Let $x$ be a relevant (with respect to the abstract properties) identifier in a boolean guard $g$, and $y$ be an identifier which is defined inside a statement $s$, depending on $g$. Then, the implicit flow from $x$ to $y$ must be added to $\Psi(s)$, yielding $\Psi(s)[x \rightsquigarrow y]$.
\[ \psi [x \to y] = [\psi]^{y_0}_{y_0} \land (y_0 \to (y'_o \lor x_i)) \]

The corresponding definitions for sets of identifiers are obtained by disjunction and iteration over \( X \) and \( Y \).

\[ \psi [X \to y] = [\psi]^{y_0}_{y_0} \land y_0 \to (x_i \lor (\forall z \in X x_i)) \]
\[ \psi [X \to \emptyset] = \psi \]
\[ \psi [X \to (\{y\} \cup Y)] = (\psi [X \to y]) [X \to Y] \]

**Definition 5.21 (Expressions).** The information flow behavior of an expression with no side effects is computed by

\[ \Psi_e (e) = \text{RET}_o \leftrightarrow \left( \bigvee \{ x_i \mid x \in \text{REF}_e (e) \} \right) \]

where RET is a special boolean variable which holds if the final value of the expression can contain secrets.

The function \( \Psi \) computes the boolean function representing the behavior of a statement with respect to abstract information flows. \( \text{REF} \) computes relevant identifiers in expressions, parameterized on the desired ANI security policy, that is, the abstract properties an observer should not be able to distinguish. \( \Psi \) analyzes the body of a method instance compositionally, starting from the type environment \( \varepsilon \) (which is left implicit).

**Definition 5.22 (Statements).** The information flow behavior of statements is computed by the function \( \Psi \). For every identifier \( y \) which is not affected by the statement, the formula \( y_0 \to y_i \) is implicitly added.

\[ \Psi (x=e) = [\psi]^{y_0}_{y_0} \land \left( \text{RET}_t \leftrightarrow \bigwedge \{ y_o \mid y \in \text{SH}_e (x) \} \right) \]
\[ \Psi (s_1; s_2) = [\Psi (s_1)]^{y_0}_{y_0} \land [\Psi (s_2)]^{y_0}_{y_0} \]
\[ \Psi (\text{if} \ (e) \ s_1 \ \text{else} \ s_2) = (\Psi (s_1) \lor \Psi (s_2)) [\text{VARS} (e) \to \text{DEF} (s_1) \cup \text{DEF} (s_2)] \]
\[ \Psi (\text{while} \ (e) \ s) = \text{LFP}_t \land \text{def} (\text{var}) \to \text{DEF} (t) \]
\[ \Psi (\nu.m.\ (e_1 \ldots e_k)) = \bigvee_{C \in T_m (\nu)} \Psi' (C.m (e_1 \ldots e_k)) \]
\[ \Psi' (C.m (e_1 \ldots e_k)) = \left[ \bigwedge_{j=1..k} \Psi_{e_j} (e_j) \right]^{\text{RET}_o}_{t} \land \left[ \Psi_{e_j} (C.m, e_1 \ldots e_k) \right]^{t}_{t} \]

- When an identifier \( x \) is assigned, the secret content of \( e \) may propagate to the sharing identifiers for \( x \). Therefore, all sharing \( y \) can contain secrets if some relevant variable of \( e \) has secrets (see Definitions 5.4 and 5.21).
5.4 Source code analysis via boolean functions

- The rule for sequential composition helps to understand subscript substitutions. The use of a fresh subscript \( t \) (which corresponds to the program point between \( s_1 \) and \( s_2 \)) creates a correspondence between the output of \( s_1 \) and the input of \( s_2 \). Therefore, \( x_o \) holds after \( s_2 \) if (i) \( x \) acquires secrets in \( s_2 \); or (ii) \( x_t \) holds after \( s_1 \) and \( x \) is not modified in \( s_2 \).
- In conditional statements, both paths must be taken into account. Moreover, implicit flows keep track of secrets in the boolean guard. Note that, in computing the relevant guard identifiers, \( \text{VAR}_e(e) \) is used instead of \( \text{REF}_e(e) \). In facts, constancy with respect to the abstract property is not enough to decide which path is to be taken at runtime (for example, in IMP, knowing that \( x \) is even gives not enough information about the test \( x = 0 \)). In terms of dependency calculus, this amounts to say that control dependencies are not abstracted.
- Computing dependencies for a loop involves a fixpoint computation. A loop is equivalent to every possible sequential composition of the statement if \( (e) \) s else skip. Standard fixpoint approximation techniques, such as widening (Section 3.1.5), can be used in order to ensure that the fixpoint computation will terminate.
- When a method is invoked, the analyzer must consider every method instance which is compatible with the actual parameters. This is done by the disjunction on \( T_e(v) \) and by computing \( \Psi^e(C.m, \ldots) \). Actual parameters are related to formal parameters by replacing \( \text{RET}_o \) with the corresponding input instance of the formal parameter \( p^i \) in \( \Psi_e(e^i) \).

**Static members**

Definition 5.22 can account for static fields by simply extending the definition of \( \text{SH} \). Static fields are guaranteed to be separated with regard to their set of locations, so that every class has its own place in the heap.

\[
\text{SH}_e(C.f) = \{C.f\}
\]

Boolean functions for static methods are computed by \( \Psi' \), which can also be seen as a one-class restriction of the method invocation \( \Psi \) rule.

5.4.2 Whole-program analysis

It is easy to see that the computation of \( \Psi \) for a method instance can lead to problems, due to mutual recursion [68]. In facts, the information flow behavior of \( d \) depends on the behavior of every \( d' \) which is invoked inside \( d \). Since \( d' \) can in turn invoke \( d \), the termination of the \( \Psi \) computation is not guaranteed. To avoid this problem, we perform a fixpoint computation on \( \psi \)-states \( \psi \in F^\lambda_R \), that is, functions from method instances to boolean functions.

\[
\psi \in F^\lambda_R = [M \mapsto F_R]
\]
A special $\psi$-state is

$$\Psi_\perp = \lambda d. \psi_\perp$$
$$\psi_\perp = \bigwedge_x x_o \leftrightarrow x_i$$

in which every method instance is neutral (that is, equivalent to skip) with respect to information flow, leaving unchanged all identifiers.

We show a technique which is quite a standard fixpoint computation on $\psi$-states. Let $\Psi_\perp$ be the initial state. When a method is invoked, its instances are pushed on the method stack, which initially contains the (unique) instance of main(). At each step, the method instance $d$ on top of the stack is analyzed. To avoid problems related to mutual recursion, when $m()$ is invoked inside $d$, every $d' \in \text{INST}(\cdot, m, \cdot)$ is pushed on the stack and analyzed if and only if it is not already somewhere on the stack. Otherwise, the current boolean functions $\Psi(d')$ is taken. When the analysis of $d$ is complete, $\Psi(d)$ is updated with the new computed boolean function.

In this new formulation, the definition of $\Psi^\varnothing(C.m, e_1 \ldots e_k)$ must be restated:

$$\Psi^\varnothing(C.m, e_1 \ldots e_k) = \bigwedge \{ d_\varnothing \mid d \in \text{INST}_\varnothing(C.m, e_1 \ldots e_k) \}$$
$$d_\varnothing = (d \text{ is on the stack}) \land (\Psi(d)) ; (\Psi(d))$$

where $\Psi$ is the current $\psi$-state. In the following algorithm, main is the method instance of main().

```plaintext
ANI {
    foreach $d$. $\Psi(d) \leftarrow \psi_\perp$
    do {
        $\Psi \leftarrow \text{analyze}(\text{main}, \Psi)$
    } until (no changes occur to $\Psi$
             or $n$ iterations have been done)
    return $\Psi$
}

analyze($d$, $\Psi$) {
    push($d$);
    foreach method call {
        foreach compatible method instance $d'$ {
            if ($d'$ is on the stack) {
                $d'_\varnothing \leftarrow \Psi(d')$
            } else {
                $d'_\varnothing \leftarrow \text{analyze}(d', \Psi)$
            }
        }
    }
}
```
5.4 Source code analysis via boolean functions

// $d'_\varphi$ is used to compute the boolean function
// for the method invocation

...

// analysis of other statements
...

// the analysis of $d$ yields a boolean function $\psi$
$\Psi(d) \leftarrow \psi$;
pop($d$);
}

The algorithm checks method instances until the stack is empty, that is, until the analysis of main is complete.

5.4.3 Widening

The ANI procedure terminates when a fixpoint is reached or, in alternative, the number of performed iterations reaches a fixed threshold $n$. In this case, the problem arises of how to deal with final $\Psi$-states which are not fixpoints.

Simply taking the $n$-th $\Psi$-state as the final result of ANI could result in an incorrect behavior of the analyzer. In facts, the algorithm starts under the optimistic assumption that no dangerous flows exist, being the behavior of methods neutral with respect to Non-Interference (which is the meaning of $\psi_\bot$). If the computation is artificially stopped before reaching the fixpoint, then we are not guaranteed that data propagation has been completely taken into account. Consequently, programs could be possibly accepted even if some dangerous flow may exist. The Abstract Interpretation theory uses widening operators (Section 3.1.5) in order to enforce soundness in bounded fixpoint computations.

In our framework, the widening of a $\Psi$-state $\Psi$ yields boolean functions $\Psi(d)$ which are correct with respect to the real information flow descriptions of methods, that is, which are more pessimistic with respect to the satisfiability of the security requirements. The most naive choice is to set $\Psi(d)$ to $\psi_\top$ for every $d$. The formula $\psi_\top$ is one which is maximal with respect to the ordering on boolean functions (introduced in Section 5.5.1); for example, TRUE can be such a formula. Obviously, this choice would lead to the rejecting of every possible program. A smarter solution is to define the widening operator as follows:

**Definition 5.23 (Finished method instances).** When executing `analyze()` on $d$ and $\Psi$, we must keep record of when the boolean function for $d$ has reached a fixpoint. For example, if $d$ contains no method invocations, then its final information flow behavior can be found at the first iteration of the ANI() main loop. We need to keep a boolean value for every instance; this
value can be updated (from false to true) at the end of analyze(), based on the following rule:

\[
\text{FINISHED}(d) = \forall d' \text{ invoked inside } d. \text{ FINISHED}(d')
\]

(clearly, the predicate is trivially true if no invocations occur in \(d\)). This means that \(d\) can be labeled as finished if every method invocation \(d'\) inside \(d\) has been previously labeled as finished.

**Definition 5.24 (Widening).** The widening operator \(\text{widening} : \mathcal{F}_B^\lambda \rightarrow \mathcal{F}_B^\lambda\) is an extensive operator on \(\psi\)-states. A further iteration of the main loop is performed, in which recursive calls are replaced by (i) if its analysis is finished: the current value for \(d'\); (ii) otherwise: a safe, pessimistic boolean function \(\psi_\top\).

\[
\text{WIDENING}(\Psi) = \forall d. \text{ analyze } d \text{ with:}
\]

\[
\forall d' \text{ invoked in } d. \ (\text{FINISHED}(d')) \ ? \ (\Psi(d')) \ ; \ (\psi_\top)
\]

**Remark 5.25.** Actually, \(\text{widening}\) is not properly a widening operator, as defined in Definition 3.3. To fit the definition, we define \(\vee_n : \mathcal{F}_B^\lambda \times \mathcal{F}_B^\lambda \rightarrow \mathcal{F}_B^\lambda\) as operating on the ascending chain \(x^0, x^1, \ldots, x^m, \ldots\) and on the bound \(n\).

\[
\begin{align*}
y^0 &= x^0 \\
y^{m+1} &= x^{m+1} & \text{if } m < n \\
y^{m+1} &= y^m \vee_n x^{m+1} = \text{widening}(x^{m+1}) & \text{otherwise}
\end{align*}
\]

It is easy to verify that this definition of the \(y^m\) chain satisfies the widening conditions.

**Example.** Let a program \(P\) be characterized by the below graph, representing the following relation: \(d \rightarrow d'\) if and only if the method instance \(d'\) in invoked inside \(d\).

\[
\begin{align*}
d & \rightarrow d' \\
& \rightarrow d'' \\
da_1 & \rightarrow d''_1 \\
da_2 & \rightarrow d''_2
\end{align*}
\]

After the first iteration of ANI(), only \(d''\) is labeled as finished. The other method instances cannot be assigned the FINISHED label because of the mutual recursion between \(d''_1\) and \(d''_2\).

**5.5 Correctness and further discussion**

This section outlines a correctness proof for the ANI() procedure, relying on the ordering on \(\psi\)-states and the monotonicity of a single iteration of the main loop.
5.5 Correctness and further discussion

5.5.1 The ordering on $\psi$-states

In order to guarantee that a fixpoint on $\psi$-states will eventually be reached, we must prove that a single iteration of the ANI() procedure is monotone on $\psi$-states.

**Definition 5.26 (Ordering on $\psi$-states).** The ordering on $\psi$-states is defined as follows:

\[
\psi_i = \left( \bigwedge_{x \in L} \neg x_i \right) \land \left( \bigwedge_{y \in H} y_i \right)
\]

\[
\psi' \leq \psi'' \equiv \forall d. \, \psi'(d) \leq \psi''(d)
\]

\[
\psi' \leq \psi'' \equiv [\psi_i \land \psi'] \subseteq [\psi_i \land \psi'']
\]

Among the reasonable boolean functions (that is, those which are a description of some legal program), the formula $\psi_\bot$ is minimal with respect to this ordering, because the only model of $\psi_i \land \psi_\bot$ is $\{y_i | y \in H\} \cup \{y_o | y \in H\}$.

Since the number of identifiers is finite, the number of models is also finite (the cardinality of the powerset of boolean variables). Consequently, there does not exist any infinite ascending chain $\psi_0, \ldots, \psi_k, \ldots$; if monotonicity can be proven, the fixpoint computation is guaranteed to terminate in a finite number of steps. The WIDENING operator of Section 5.4.3 results to affect the efficiency, rather than the termination of the algorithm.

5.5.2 Monotonicity

Proving monotonicity for an iteration of ANI() involves checking the monotonicity of the analyze() procedure. This can be done by demonstrating that all rules of $\Psi$ are monotone on their parameters. Some easy lemmata on the boolean function ordering are provided.

**Lemma 5.27.** Conjunction preserves monotonicity:

\[
\psi'_1 \leq \psi''_1 \land \psi'_2 \leq \psi''_2 \implies (\psi'_1 \land \psi'_2) \leq (\psi''_1 \land \psi''_2)
\]

Proof. Trivial.

**Lemma 5.28.** Disjunction preserves monotonicity:

\[
\psi'_1 \leq \psi''_1 \land \psi'_2 \leq \psi''_2 \implies (\psi'_1 \lor \psi'_2) \leq (\psi''_1 \lor \psi''_2)
\]

Proof. Trivial.
Lemma 5.29. Subscript substitution preserves monotonicity:

\[ \psi' \leq \psi'' \implies [\psi']_u \leq [\psi'']_u \]

Proof. Trivial (subscript substitution does not alter the nature of boolean functions).

Lemma 5.30. Let \( \mathcal{M} \) be a variable assignment; \([\mathcal{M}]_{y_t}^{x_s}\) is defined to be such that \(x_s \in [\mathcal{M}]_{y_t}^{x_s} \iff y_t \in \mathcal{M}\) and, for every \(z_u \neq x_s, z_u \in [\mathcal{M}]_{y_t}^{x_s} \iff z_u \in \mathcal{M}\). Then, for every \(\psi\), we have

\[ \mathcal{M} \in \left[ [\psi]_{y_t}^{x_s} \right] \iff [\mathcal{M}]_{y_t}^{x_s} \in \left[ \psi \right] \]

Proof. \(y_t\) occurs in \([\psi]_{y_t}^{x_s}\) where (i) it occurs in \(\psi\); and (ii) \(x_s\) occurs in the same place in \(\psi\). If \(\mathcal{M}\) is a model of \([\psi]_{y_t}^{x_s}\), then there exists a model \(\mathcal{M}'\) of \(\psi\) (which is equal to \(\mathcal{M}\) for every variable which is not \(x_s\) or \(y_t\)) such that \(x_s \in \mathcal{M}' \iff y_t \in \mathcal{M}'\), since \(x_s\) and \(y_t\) are considered as a single variable in \([\psi]_{y_t}^{x_s}\). Moreover, by stating \(y_t \in \mathcal{M}' \iff y_t \in \mathcal{M}\), we have \(\mathcal{M} = [\mathcal{M}]_{y_t}^{x_s}\). The other direction of the implication is similar.

Lemma 5.31. Variable substitution preserves monotonicity, under the hypothesis that the replaced variable is not the input instance of any \(x\):

\[ \forall \psi', \psi'', x_s, y_t. \ s \neq i \land \psi' \leq \psi'' \implies [\psi']_{y_t}^{x_s} \leq [\psi'']_{y_t}^{x_s} \]

Proof. The condition on \(x_s\) implies \([\psi]_{y_t}^{x_s} = \psi_1\). Then

\[ \mathcal{M} \in \left[ \psi_1 \wedge [\psi']_{y_t}^{x_s} \right] \iff [\text{Lemma 5.30}] \]

\[ [\mathcal{M}]_{y_t}^{x_s} \in \left[ \psi_1 \wedge [\psi]_{y_t}^{x_s} \right] \iff [\text{hypothesis}] \]

\[ [\mathcal{M}]_{y_t}^{x_s} \in \left[ \psi_1 \wedge [\psi'']_{y_t}^{x_s} \right] \iff [\text{hypothesis}] \]

\[ \mathcal{M} \in \left[ \psi_1 \wedge [\psi'']_{y_t}^{x_s} \right] \]

The hypothesis \(s \neq i\) is not restrictive because no substitution \([\psi]_{z_t}^{x_s}\) occurs in the definition of \(\Psi\).

Lemma 5.32. Adding implicit flows to boolean functions preserves monotonicity:

\[ \forall X, Y. \ \psi' \leq \psi'' \implies \psi' [X \rightarrow Y] \leq \psi'' [X \rightarrow Y] \]
Proof. First, we prove that, for a set $X$ and an identifier $y$,
$$
\psi' \leq \psi'' \implies \psi' [X \rightsquigarrow y] \leq \psi'' [X \rightsquigarrow y]
$$
This is true since
$$
\psi' \leq \psi'' \implies [\text{Lemma 5.31}]
$$
$$
[\psi']^o_{y^o} \leq [\psi'']^o_{y^o} \implies [\text{Lemma 5.27}]
$$

\[
\left( [\psi']^o_{y^o} \land (y_o \leftarrow y'_o \lor (\forall x \in X \times i)) \right) \leq
\leq \left( [\psi'']^o_{y^o} \land (y_o \leftarrow y'_o \lor (\forall x \in X \times i)) \right) \implies [\text{Def. 5.20}]
\]
$$
$$
\psi' [X \rightsquigarrow y] \leq \psi'' [X \rightsquigarrow y]
$$
The general case $\psi' [X \rightsquigarrow Y] \leq \psi'' [X \rightsquigarrow Y]$ can be obtained by iterating the above derivation for every $y \in Y$.

**Definition 5.33.** The function $\Psi [s]$ is a specialized version of $\Psi$, parameterized on a statement $s$. The parameters for $\Psi [s]$ replace the recursive calls to $\Psi$ or $\Psi_s$ inside the corresponding definition of $\Psi$.

$$
\Psi [s] = \lambda \psi. \ [\psi]^o \land \left( \text{RET}_o \leftrightarrow \bigwedge \{ y_o \mid y \in \text{SH}_e (x) \} \right)
$$
$$
\Psi [s_1 : s_2] = \lambda \psi_1. \lambda \psi_2. \ [\psi_1]^o \land [\psi_2]^i
$$
$$
\Psi [\text{if} (e) \ s_1 \ \text{else} \ s_2] = \lambda \psi_1. \lambda \psi_2. \ (\psi_1 \lor \psi_2) [X \rightsquigarrow Y]
$$
$$
X = \text{REF}_e (e)
$$
$$
Y = \text{DEF} (s_1) \cup \text{DEF} (s_2)
$$
$$
\Psi [\text{while} (e) \ s] = \lambda \psi. \ \text{LFP}_{\land \psi_1, \psi_2, \ x, i} \left( \lambda \phi. \ \psi \lor \left( \left[ \psi \right] \land \left[ \psi \right] ^i \right) \right)
$$
$$
\Psi [\text{return} \ e] = \lambda \psi. \ [\psi]^o \land \text{RET}_o \leftrightarrow \text{RET}_o
$$

Actually, rather than a statement, $s$ is for a statement schema: for example, if $(e) \ s_1 \ \text{else} \ s_2$ stands for an arbitrary conditional statement.

**Theorem 5.34.** For every statement schema $s$, $\Psi [s]$ is monotone on its parameters. That is, for $\Psi [s]$ with $k$-arity,

$$
\forall j \in \{1, \ldots, k\}. \ \psi_j' \leq \psi_j'' \implies \Psi [s] (\psi_1', \ldots, \psi_k') \leq \Psi [s] (\psi_1'', \ldots, \psi_k'')
$$

Proof. By cases:
• Case (5.1): follows by Lemmata 5.31 and 5.28.
• Case (5.2): follows by Lemmata 5.27 and 5.29.
• Case (5.3): $X$ and $Y$ can be computed once for given $c$, $s_1$ and $s_2$; then, they can be regarded as constant sets. The result follows by Lemmata 5.32 and 5.28.
• Case (5.4): the function $\lambda \phi. \psi \lor \left( [\phi]^c \land [\psi]^c \right)$ is monotone on $\phi$ by Lemmata 5.31, 5.27, and 5.28; then, finding its fixpoint is meaningful. It is also monotone on $\psi$, by the same lemmata. Consequently, the lfp is monotone on $\psi$.
• Case (5.5): the parameter $\psi$ is built out of all possible instances of $m$ that can be called by $v$; the result follows by Lemmata 5.31 and 5.27.
• Case (5.6): follows by Lemmata 5.31 and 5.27.

\[ \\]

**Theorem 5.35.** For any method instance $d$, the procedure $\text{analyze}(d, \Psi)$ is monotone on $\Psi$.

**Proof.** Monotonicity for non-invoking statements follows directly by Theorem 5.34. For each method invocation, the set of compatible method instances does not depend on $\Psi$. The boolean function for $d'$ is either directly taken from $\Psi$ or computed by $\text{analyze}()$ on $d'$ and $\Psi$. In both cases, monotonicity on $\Psi$ holds. $d'_o$ is computed for every $d'$; the monotonicity of the disjunction follows by Lemma 5.28. \[ \Box \]

Theorems 5.34 and 5.35 prove that the ANI procedure indeed reaches a fixpoint which describes the information flow of a program. Since no infinite ascending chains exist, termination is guaranteed. However, a widening operator was defined (Section 5.4.3) in order to improve efficiency.

### 5.6 The link to Program Slicing

In section 3.4.2, the strong connection between Program Slicing and Information Flow was outlined. A quite natural extension of standard slicing consists of considering abstract properties instead of concrete properties: the statements which are relevant to a criterion $\langle j, v \rangle$ (and, therefore, which should be included in the abstract slice) are those which can affect the abstract value of $v$ at the program point $j$. Rival [65] includes an abstract version of program slicing, which still lacks a comprehensive definition\(^2\), among the applications of the abstract dependency calculus.

\(^2\) Despite its title, the recent work by Hong, Lee and Sokolsky [44] does not discuss abstract slicing in the meaning we are investigating.
5.6.1 Computing abstract slices

Consider the simple slicing algorithm in Section 3.4.1. It iteratively computes the functions $R^k_S(i)$ (for a statement at program point $i$), $S^k_S$ and $B^k_S$ for a slicing criterion $S$, starting with the index $k = 0$. These functions provide the solution of dataflow equations for computing the set of statements which are relevant to $S$. They rely on $\text{ref}()$ and $\text{def}()$, that is, respectively, the set of referred and defined variables in a statement. The main difference between the concrete and the abstract case can be encapsulated into the $\text{ref}()$ function, which captures data dependencies\footnote{As explained in Definition 5.22, control dependencies are not affected by the property abstraction.} (Section 5.2.2). Given an abstract domain $\rho$ which describes the abstract properties we want to study, the following definition of $\text{ref}()$ can be given:

\[
\text{ref}(v := e) = \text{abstract, parameterized on } \rho
\]
\[
\text{ref}(\text{if } e \text{ then } \cdot \text{ else } \cdot) = \text{vars}(e)
\]
\[
\text{ref}(\text{while } e \text{ do } \cdot) = \text{vars}(e)
\]

By simply replacing the abstract version of $\text{ref}$, parameterized on an abstract property $\rho$, in the dataflow equations computing $B$, $R()$ and $S$, the computed slice turns out to be smaller than the concrete one.

Example. Consider the program $P'$ (the same as Example 3.4.2), in Figure 5.2. If the sign property is considered, then the final value of $l_1$ does not depend on the initial value of $h_1$ since:

- before entering the loop, $l_1$ is always positive, due to the assignment $l_1 := 1$;
- adding 1 to a positive number does not change its sign.

---

**Fig. 5.2.** The program $P'$

\begin{verbatim}
  h1 = h1;                      1
  h2 = h2;                      2
  l1 = 1;                       3
  while (h1 > 0) do {
    l1 = l1 + 1;                4
    h1 = h1 - 1;                5
  }                             6
  l1 = l1;                     7
  l2 = l2;                     8

\end{verbatim}
Consequently, the statement $h_1 := h_1$ should not be included in the abstract slice. The behavior of the slicing algorithm should be consistent with these requirements. The statement $h_1 := h_1$ is not included in the abstract slice because, when $l_1 := l_1$ is executed, the value of $l_1$ is constant with respect to the abstract property. This result is obtained by the type analysis of the program, which yields $\varepsilon_{s}(l_1) = [\text{pos}]$, so that $\text{ref}_{\varepsilon_{s}}(l_1) = \emptyset$. Therefore, the loop does not change the sign of $l_1$ and its guard is not among the relevant variables.

**Proposition 5.36.** Let $\rho'$ and $\rho''$ be two abstract domains, with $\rho'$ more precise than $\rho''$. Given a program $P$ and a slicing criterion $S$, the abstract criteria $S'$ and $S''$ are its reformulation on, respectively, $\rho'$ and $\rho''$. Then, the abstract slice with respect to $S''$ is smaller than the slice on $S'$.

**Proof.** By Proposition 5.7, we have, for every expression $e$, $\text{ref}_{\rho'}(e) \subseteq \text{ref}_{\rho''}(e)$. Both $R_0^S(i)$ and $S_0^S$ are monotone on $\text{ref}_S$, then, for every program point $i$,

$$R_0^{S'}(i) \subseteq R_0^S(i) \quad S_0^{S'} \subseteq S_0^S.$$  

This result can be rewritten as $R[0] \wedge S[0]$, given

$$R[k] \equiv R_{S'}^{k}(i) \subseteq R_{S}^{k}(i)$$

$$S[k] \equiv S_{S'}^{k} \subseteq S_{S}^{k}$$

$$B[k] \equiv B_{S'}^{k} \subseteq B_{S}^{k}$$

Moreover, the following easy inductive steps hold for every $k \geq 0$:

$$S[k] \Rightarrow B[k]$$

$$R[k] \wedge B[k] \Rightarrow R[k+1]$$

$$B[k] \wedge R[k+1] \Rightarrow S[k+1]$$

Consequently, $S[k]$ holds for every $k$, and, in particular, for the fixpoint of the iteration sequence. 

**5.6.2 Obtaining consistent slices**

An important requirement for a slice is to be a correct program. In particular, we may ask for all variables to be initialized before being used.

**Example.** Consider the assignment $s : x := e[y, z]$: data are consistent if $y$ and $z$ are initialized in every path from the first statement to $s$. We assume this condition holds for $P$. In Weiser algorithm, a slice for $<s, x>$ contains all statements $s'$ such that $y$ or $z$ belong to $\text{def}(s')$, and there is at least one path reaching $s$ where the effects of $s'$ are preserved. Consequently, the slice
$P_{(s,x)}$ is guaranteed to contain all useful initializing statements for $y$ and $z$, then it is correct with respect to variable initialization.

In abstract slicing, this kind of consistency is no longer preserved: statements $s'$ of the previous example, defining $y$ (resp. $z$) can be deleted from the slice even if they are not shadowed, provided that there is no abstract dependency of $e$ on $y$ (resp. $z$).\footnote{The same problem would arise if concrete dependencies were computed semantically. For example, in the program $x := y ; v := x - x$, the variable $v$ does not depend, semantically, on $x$, then the first statement should be deleted from the slice.} Then, $e [y, z]$ may have some of its variables uninitialized. To deal with this problem, we observe that our algorithm $\text{REF} (e)$ in Section 5.2 computes, together with the set $X$ of relevant variables, a simplified expression $e'$ which rules out variables occurring in constant-equivalent subexpressions. Then, assignments $x := e$ could be replaced, in the slice, by $x := e'$. Unfortunately, as pointed out after Proposition 5.13, it is not guaranteed that $y \in \text{VARS} (e') \Leftrightarrow y \in X$; consequently, it could happen that some variables in $e'$ lack initialization. In order to compute a consistent slice, we could take $\text{VARS} (e')$ (instead of $X$) as the set of relevant variables (thus obtaining a less precise slice). However, future research will be directed to modify $\text{REF}$ in order to obtain $y \in \text{VARS} (e') \Leftrightarrow y \in X$.

5.6.3 Discussion

Abstract program slicing is an interesting application of the abstract dependency calculus to standard program slicing. A formal and complete definition of what abstract slicing should do is still to be provided. Future research should study the necessary and sufficient conditions for having a semantically correct abstract slice of a program. Correctness conditions should help in the designing of an algorithm which is able to keep irrelevant statements if they are necessary for the slice correctness, and to modify statements in order to eliminate undesired syntactic (concrete) dependencies.

Moreover, the same techniques could be applied to more advanced versions of slicing [69], such as dynamic slicing [47] or conditioned slicing [13].
No one can build his security upon the nobleness of another person.

Willa Cather (1873 - 1947)

I love quotations because it is a joy to find thoughts one might have, beautifully expressed with much authority by someone recognized wiser than oneself.

Marlene Dietrich (1901 - 1992)

This chapter discusses how to apply the Information Flow analysis outlined in Chapter 5 to code certification. An architecture in the style of Proof-Carrying code (Section 3.5) is introduced. In order to be executed by mistrustful users (that is, users which are not a priori confident in the safety of programs), programs must be supplied together with a correctness proof for the desired security policy: Abstract Non-Interference.

Security checking of source code programs is a valid tool for programmers to design safe software. Yet, in general, programs are sold or provided in their executable form. Therefore, an external user which wants to safely run a program will reasonably require more than a correctness proof of the source code, since he or she is not guaranteed that the security properties of the source program are preserved in the executable code.

Since we are dealing with Java-like JL programs, software is not supplied to users as native, machine-dependent code. Instead, Java source code is compiled to bytecode (Section 2.3), which is executed by the Java Virtual Machine (JVM [52]) interpreter. The problem is to verify that bytecode programs (coming as sets of .class files) indeed satisfy the security requirements. The first step is to define an algorithm for checking ANI on bytecode.
6.1 Analyzing bytecode programs

The syntax and semantics of bytecode are described in Section 2.3. It should be clear that bytecode, although designed as the target language of Java compilers, is not forced to be produced from Java source code. The Java Virtual Machine knows nothing of the Java programming language, only of a particular binary format: the .class file format. Bytecode files can be also obtained from other programming languages or techniques; the JVM does not need to know the origin of .class files. The possibility to use bytecode in non-Java programming frameworks considerably increases the range of application for this security analysis.

6.1.1 Bytecode boolean functions

Boolean functions will be used to describe the information flow behavior of bytecode programs, similarly to their rôle in source code analysis. Before describing how boolean functions are computed, we introduce some new boolean variables for describing semantic objects\(^1\) taking part to program execution.

Boolean variables can represent field references (that is, a subset of the elements \(R[j]\) for a Runtime Constant Pool \(R \in R\)); operand stack values (that is, the \(k\)-th element \(O[k]\) in some operand stack \(O \in O\)); local variables (that is, \(X[l]\) for \(X \in X_{loc}\)). Given an operand stack \(O\) and an instruction \(i\), \(\tau_i\) and \(\tau_o\) are, respectively, the index of the top element of the stack before and after \(i\). The variables \(O[j]_i\) and \(O[j]_o\) denote the input and output instances of the value at position \(j\) on the stack. Typically, \(O[\tau_i]_i\) and \(O[\tau_o]_o\) represent, respectively, the boolean variables corresponding to the top of the stack on input and output. The elements under the top of the stack are denoted by \(O[t:k]\) for some \(k > 0\). Substitutions \([\psi]_t^i\) and \([\psi]_t^o\) are defined as in source code analysis (Definition 5.17).

**Definition 6.1.** When two formulae \(\psi\) and \(\psi'\) are sequentially combined, some of the output variables of \(\psi\) need to have their subscript replaced by a fresh \(t\).

The operator \(\cdot\) encodes the concatenation of boolean formulae:

\[
\psi \cdot \psi' = [\psi]_t^i \land \psi' \land \bigwedge_{v_t \notin vars(\psi')} v_t \leftrightarrow v_o
\]

The equivalences \(v_t \leftrightarrow v_o\) are added for variables which are not affected by \(\psi'\).

The substitution \([\psi']_t^o\) is not necessary since it is explicit in the rules below.

6.1.2 A simple source code compiler

A naïf, non-optimized compiler is presented. The process of compilation is schematized as \(C(s^S) = s^B\), where \(s^S\) is a source code statement and \(s^B\) is

\(^1\) Here, the word object does not refer, specifically, to a class instance.
the result of compiling $s_S$. In the rules, $j_v$ stands for the index of $v$ in the set of local variables $X$; $h_y$ stands for the index of a reference to $y$ (which can be, for example, a class or a method) in the Runtime Constant Pool $R$.

\[
C(v) = \text{aload } j_v \\
C(\text{new } C(e^1 \ldots e^k)) = \text{new } r_C; \text{dup;} C(e^1); \ldots; C(e^k); \\
\text{invokespecial } r_C, \text{init} \\
C(v.f) = C(v); \text{getfield } r_f \\
C(v.m(e^1 \ldots e^k)) = C(v); C(e^1); \ldots; C(e^k); \text{invokevirtual } r_m \\
C(v=e) = C(e); \text{astore } j_v \\
C(v.f=\epsilon) = C(v); C(\epsilon); \text{putfield } r_f \\
C(s_1; s_2) = C(s_1); C(s_2) \\
C(\text{if } (e'==\epsilon) \ s_1 \text{ else } s_2) = C(e'); C(\epsilon'); \text{ifacmpneq } p; C(s_1); \\
goto q; p: C(s_2); q: \ldots \\
C(\text{while } (e'==\epsilon'') \ s) = p: C(e'); C(\epsilon''); \text{ifacmpneq } q; \\
C(s); \text{goto } p; q: \ldots
\]

6.1.3 The Control Flow Graph

In a bytecode program, statements are not organized in a block structure (as in while $(e) \{\ldots\}$). Instead, the control flow is regulated by labels and jumps. Consequently, program analysis cannot be purely compositional.

The instruction sequence of a method body can be schematized in a Control Flow Graph (CFG), in which blocks of sequential instructions are linked by jump edges. Bytecode jump instructions are goto (unconditional jump), ifacmpneq (comparison for equality on a reference value), ifnull (comparison for equality with the null value), ificmpeq (comparison for equality on integers), and ifeq (comparison for equality with zero). We ignore the last two instructions because we are not interested in primitive types.

CFG jump edges

Every bytecode instruction $\iota$ has at most two successors, that is, instructions which can be possibly executed after $\iota$. For non-jump instructions, the successor of $\iota$ is the instruction directly following it in the code. Control flow must be represented as a graph having, as vertexes, sequence of non-jump instructions (blocks), and, as edges, the elements of the successor relation induced by jumps. A block $b$ can end in a non-jump instruction $\iota$ (for example, if the successor of $\iota$ is the target of some jump) or in a jump. If $\iota$ is non-jump or goto, then $b$ has one successor block; otherwise, $b$ has two successors (the branches of the conditional jump).
Example. Let the block structure of a program $\mathcal{P}$ be represented by the Control Flow Graph in Figure 6.1. In this graph, the most common block structures are exemplified.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig61.png}
\caption{An example of CFG}
\end{figure}

Implicit flows

An edge $\omega$ from a block $b$ to $b'$ (written $b \xrightarrow{\omega} b'$) is labeled with the set $\delta_\omega$ of identifiers which can affect the execution of $b'$. Intuitively, sets $\delta_\omega$ play the role of $\text{REF}_e(e)$ in the definition of $\Psi$ (Definition 5.22) for the conditional statement. For example, if, at the end of $b$, execution branches on $x$, then $x$ belongs to the $\delta$ labels of both edges exiting from $b$, possibly together with identifiers inherited from higher-level (with respect to the program structure) conditional jumps.

More formally: let $b$ be a block ending in $\iota$. $b$ inherits from its predecessor blocks a set $\delta_b$ of identifiers, defined as: $\cup\{\delta_\omega' \mid b' \xrightarrow{\omega'} b\}$; it is the set of identifiers whose value can possibly determine whether $b$ will be actually executed. Sets of branching identifiers for $b$ are computed as follows, depending on $\iota$.

- $\iota$ is not a jump: $b$ has an only outgoing edge $\omega$ with $\delta_\omega = \delta_b$.
- $\iota = \text{goto}$: $b$ has an only outgoing edge $\omega$ with $\delta_\omega = \delta_b$ (no additional branching can be done).
- $\iota = \text{if\_acmpeq}$: there are two outgoing edges $\omega'$ and $\omega''$, with $\delta_\omega' = \delta_\omega'' = \delta_b \cup \{O[\tau_i], O[\tau_i - 1]\}$, where $O$ is the operand stack when executing $\iota$. This means that branching on comparison depends on both compared values.
• \( \tau = \text{ifnull} \) : both outgoing edges are labeled with \( \delta = \delta_0 \cup \{O_{\tau_1}\} \) (branching only depends on one identifier).

The initial block of a method instance \( d \) (that is, the first one to be executed) is also equipped with a set \( \delta \) of branching identifiers: in facts, the execution of \( d \) could be conditioned by control flow in the method \( d' \) which invokes \( d \).

Example. Edges in the block structure of Example 6.1.3 are labeled with branching variables as shown in Figure 6.2. The sets \( \delta' \) and \( \delta'' \) are the set of relevant variables for the boolean guards.

\[ \text{Fig. 6.2. Branching variables} \]

Control Flow Graph analysis

Let \( \Psi_b (\circ) \) the boolean function computed for a block \( b \), as defined in Section 6.1.4. \( \Psi_b (\psi_0(b)) \) is obtained by applying the abstract semantics for the instruction sequence of \( b \), starting by

\[ \psi_0(b) = \bigvee_{\{b' \mid b' \succeq b\}} \Psi_{b'} (\circ) \]

where \( \circ \) is the initial boolean function for \( b' \). The meaning of this formula is that \( b \) can be executed after one of its predecessors, then it cannot be decided statically which boolean function has to be taken. Computing \( \Psi \) obviously involves a fixpoint computation; then, a suitable technique is needed in order to resolve cycles in the successor relation.
6.1.4 Analysis of blocks

Since exceptions are not considered, instructions inside a block are always executed sequentially.

The abstract semantics

Instructions inside a block $b$ are considered sequentially via the abstract semantics $[\cdot][\cdot]$, which operates on states $S_F \times F_B \times \varphi(B)$. The analysis of the block starts in a state $(F, [\psi]_B)$ where

- $F$ is the (concrete) framestack.
- $\psi$ is the initial boolean function, obtained by disjunction of functions computed in the predecessor blocks $p[b]$: $\psi = \bigvee_{b' \in p[b]} \psi_{b'}$. If no predecessor blocks exist, then $\psi = \bigwedge_{i_0} v_i \leftrightarrow v_i$.
- $\delta$ is the set of inherited branching identifiers, as defined above. It is not modified inside the block.

Concrete semantic components are only shown for clarity.

Definition 6.2. The formula $x_t \leftarrow \delta y_u$ stands for

$$x_t \leftarrow \left( y_u \lor \left( \bigvee_{i \in \delta} v_{i'} \right) \right)$$

meaning that a modified identifier $x$ depends on the assigned value (contained in $y$) and on the branching identifiers of the current block. The subscript $i'$ is used in order to preserve the input value of branching identifiers from assignments possibly occurring inside the block. At the end of the block, the boolean function $\psi$, computed after the last instruction, is modified into $\psi'$ as follows:

$$\Psi_b = \psi' = \psi \land \left( \bigwedge_{i \in \delta} v_i \leftrightarrow v_{i'} \right)$$

The following paragraphs outline the information flow analysis of bytecode non-jump instructions, divided in categories.

Instructions working on the operand stack

These instructions only modify the current operand stack by using information which is contained in the stack itself.
6.1 Analyzing bytecode programs

Load and store instructions take, as a parameter, the index of the local variable on which they work. Information flow analysis simply keeps track of the moving of data to/from local variables from/to the operand stack.

Instructions loading from/storing to local variables

Load and store instructions take, as a parameter, the index of the local variable on which they work. Information flow analysis simply keeps track of the moving of data to/from local variables from/to the operand stack.

Instructions reading or writing fields

Getting from/putting in object fields is done by resolving the information specified in the constant pool in order to have a reference to the field. Static fields do not have on the stack a reference to the object containing desired data; instead, the constant pool contains a reference to the class.

The computed boolean function keeps track of the modifications on the stack: for example, in swap, the information flow properties of stack values are swapped. When pushing the null constant, the resulting stack value cannot contain secrets. δ is not used for this category of instructions, since stack values cannot cause interference unless they are written in local variables or fields. Note that, in [dup], the stack indexes τ₀ and τ₁ are not the same: τ₁ = τ₀ - 1 since an additional value has been pushed.

Instructions reading or writing fields

Getting from/putting in object fields is done by resolving the information specified in the constant pool in order to have a reference to the field. Static fields do not have on the stack a reference to the object containing desired data; instead, the constant pool contains a reference to the class.

The computed boolean function keeps track of the modifications on the stack: for example, in swap, the information flow properties of stack values are swapped. When pushing the null constant, the resulting stack value cannot contain secrets. δ is not used for this category of instructions, since stack values cannot cause interference unless they are written in local variables or fields. Note that, in [dup], the stack indexes τ₀ and τ₁ are not the same: τ₁ = τ₀ - 1 since an additional value has been pushed.
In the put instructions, $SH$ is the set of identifiers which can be affected by a modification in $j$.

### Method invocation

In the static analysis for Abstract Non-Interference, instructions for method invocation do not really execute methods: the operand stack is modified as if the method had really been executed and the computed boolean function is the one associated with the method itself. Rules for this kind of instructions are slightly different, depending on whether the invoked method returns a value.

- **Methods returning a value:**
  
  $\psi$ is a special variable which is also used in analyzing return instructions: it indicates the top of the operand stack which is in frame directly below the current frame in the framestack. A boolean function $\Psi(d)$ is referred for every method instance $d$ which is compatible with the current method invocation. We can consider $\Psi(d)$ to be parametric on the set of branching identifiers $\delta$ (i.e. $\Psi(d) = \lambda \delta. \psi$ where $\psi \in \mathcal{F}_B$). When the method $d$ is called, the current value of $\delta$ replaces the formal parameter in $\Psi(d)$, yielding $\Psi^\delta(d)$.

  $u$ is the return value of the method, when it exists. $z$ is a special variable which is also used in analyzing return instructions: it indicates the top of the operand stack which is in frame directly below the current frame in the framestack. A boolean function $\Psi(d)$ is referred for every method instance $d$ which is compatible with the current method invocation. We can consider $\Psi(d)$ to be parametric on the set of branching identifiers $\delta$ (i.e. $\Psi(d) = \lambda \delta. \psi$ where $\psi \in \mathcal{F}_B$). When the method $d$ is called, the current value of $\delta$ replaces the formal parameter in $\Psi(d)$, yielding $\Psi^\delta(d)$.

### Remark 6.3.

Substitution of $\delta$ inside $\Psi(d)$ is performed only once. That is, further method invocations inside $d$ are not taken into account. This avoids termination problems due to recursion.

**Example.** Let $\Psi(d)$ be $\psi' \land \psi''(\Psi(d'))$, where $\psi''(\Psi(d'))$ means that $\psi''$ depends on the boolean function of another method $d'$. Then, $\Psi^\delta(d)$ is obtained by
replacing placeholders for branching variables with \( \delta \). This substitution is only syntactic and is not performed recursively on \( \Psi(d') \).

\[ \square \]

**Instructions returning from method execution**

Rules for return instructions simply keep the security value on top of the operand stack and put it on top of the stack of the underlying frame.

\[
\text{[return]} \left\langle \frac{(X,O, R)}{(X', O', R')} \psi \right\rangle_s = \langle X', O', R', [\psi \cdot \tau_o] \rangle_s
\]

\( O[\tau_i] \) is the returned value. The boolean variable \( \tau_o \) represents the top of the stack under the current frame; it is dual to \( O[\tau_i] \) in the invokevirtual and invokespecial instructions for non-void return type methods.

**Other instructions pushing values on the operand stack**

Both ldc (pushing a value from the runtime constant pool) and new (creating a new object reference) instructions push a value on the operand stack. In the first case, the pushed value does not contain secrets, since it is constant with respect to types. In the second case, the value is secret-free if its class is ground (that is, has no subclasses).

\[
\text{[ldc]} \left\langle \frac{(X,O, R)}{(X', O', R')} \psi \right\rangle_s = \langle X, R[j] : O, R', [\psi \cdot \neg O[\tau_o]] \rangle_s
\]

\[
\text{[new]} \left\langle \frac{(X,O, R)}{(X', O', R')} \psi \right\rangle_s = \langle X, o : O, R, [\psi \cdot A] \rangle_s
\]

\( A \equiv \neg O[\tau_o] \) the new object has a ground class

\( A \equiv O[\tau_o] \) otherwise

### 6.1.5 Expressions

Our information flow analysis of source code relies on the function \( \text{ref}(e) \), which computes the set of relevant identifiers with respect to the abstract value of \( e \) (Section 5.2). We assume there is a similar function for bytecode, which uses local variables and field references instead of source code identifier.

Compiling an assignment \( v = e \) results in the bytecode instruction sequence

\[ C(e); \text{ astore } j_v \]

where \( j_v \) is the index of \( v \) in the set of local variables (compiling a field assignment \( v . f = e \) is similar). Immediately before the store instruction, the computed boolean function has the form
\[
\psi = [\psi_0]^0 \land [\Psi_e(e)]^1 \\
\Psi_e(e) = O[\tau_0]\_o \leftrightarrow \bigvee \{x_i \mid x \in \text{REF}_o(e)\}
\]

In the above formulas, \(\psi_0\) and \(\varepsilon_0\) are, respectively, the boolean function and the type environment computed before the assignment. The boolean variable \(O[\tau_0]\_o\) corresponds to the result value of \(e\). It has the same rôle as \(\text{RET}_o\) in source code analysis. The definition of \(\Psi_e(e)\) is clearly equivalent to its source code counterpart.

### 6.1.6 Program analysis

A bytecode program comes as a collection of .class files, each containing a set of method instances. Program analysis for Abstract Non-Interference works in the same way as for Java source code (Section 5.4.2). At the beginning, all method instances \(d\) are assigned to the boolean function

\[
\Psi(d) = \psi_\perp = \bigwedge_{v \in B} v_o \leftrightarrow_\delta v_i
\]

so that methods are supposed to be equivalent, with respect to information flow properties, to nop. Note that every function \(\Psi(d)\) is parametric on a set \(\delta\) of branching variables (see the rules for method invocation instructions, in Section 6.1.4).

### 6.2 The relation between source code and bytecode

Under the assumption that \(\text{REF}\) is computed in the same way, the analysis of bytecode is equivalent to its source code counterpart.

**Theorem 6.4.** A source code program \(P\) is accepted, with respect to some ANI security policy, if and only if its compiled version \(C(P)\) is accepted by the bytecode analysis on the same policy.

To informally prove this, we examine three kinds of statements. We write \(b' \equiv_{S,B} b''\) when the (source code) boolean variable \(b'\) plays (informally) the same rôle as the (bytecode) boolean variable \(b''\). For example, after evaluating an expression, both \(\text{RET}_o\) and \(O[\tau_o]\_o\) represent the computed value.

#### 6.2.1 Variable assignment

The statement \(s = v = e\) is compiled to \(C(e);\text{ astore } j_v\). The \(\Psi\) function for source code computes the boolean function

\[
\psi_S = [\Psi_e(e)]^0 \land \text{RET}_1 \leftrightarrow v_o
\]
where $SH_v = \{ v \}$. On the other hand, bytecode analysis yields

$$\psi_B = \psi_0 \cdot O[\tau_t] \rightarrow_\delta X[j_v]_0$$

which is equivalent to $\psi_S$ since:

- $RET \approx_{S,B} O[\tau_t]$ and $v_0 \approx_{S,B} X[j_v]_0$;
- $[\Psi_e (e)]_t^0$ is left implicit in $\psi_B$ (it is incorporated into the initial function $\psi_0$);
- conversely, the initial boolean function is implicit in $\psi_S$;
- implicit flows $\delta$ are not accounted for in $\psi_S$ because of the compositional structure of high-level programs (they will be computed for the conditional statement surrounding $s$).

### 6.2.2 Field assignment

A field assignment $s = v.f = e$ is compiled as

$$C (v.f = e) = C (v) ; C (e) ; \text{putfield } r_f$$

The computed boolean function for source code and bytecode is

$$\psi_S = [\Psi_e (e)]_t^0 \land RET \leftrightarrow \left( \bigwedge \{ y_o | y \in SH_e (v.f) \} \right)$$

$$\psi_B = \psi_0 \cdot O[\tau_t] \rightarrow_\delta \left( \bigwedge \{ y_o | y \in SH_e (r_f, v) \} \right)$$

Similarly to local variable assignment, equivalence follows by $\approx_{S,B}$; clearly, $SH(r_f, v)$ must be consistent with $SH(v.f)$.

### 6.2.3 Conditional statement

Let $s$ be the statement

- if $(e) \, s_1$ else $s_2$

It is compiled as

$$C (s) = E_p ; C (s_1) ; \text{goto } q ; p : C (s_2) ; q : \ldots$$

where $E_p$ is the result of compiling $e$, together with the test instruction. If $e$ is false, execution continues at the statement labeled by $p$. The boolean function for $s$ is

$$(\Psi (s_1) \lor \Psi (s_2)) \left[ \text{VARS} (e) \leadsto \text{DEF} (s_1) \cup \text{DEF} (s_2) \right]$$

On the other hand, the bytecode block $b$ beginning at $q$ has, as predecessors, the blocks $b_1 = C(s_1)$ and $b_2 = C(s_2)$. The initial function $\psi_0$ of $b$ is computed as the disjunction of the final boolean functions of $b_1$ and $b_2$, which is equivalent to $\Psi (s_1) \lor \Psi (s_2)$. Implicit flows are equivalent since $b_1$ and $b_2$ take $\delta = \text{REF} (e)$ as their branching variables.
6.2.4 Non-compiled bytecode

In general, a user which wants to execute safe programs (see next section) is not supposed to believe that the bytecode form of the program is the result of a correct compiling process. As mentioned before, the JVM does not make assumptions on how .class files are produced.

Therefore, the Control Flow Graph of programs is not forced to be a straightforward translation of high-level block structure. This is not a problem since bytecode ANI analysis does not rely on such an assumption. It can deal with arbitrary, meaningful bytecode programs, which are a superset of programs obtained by compiling.

6.3 Towards a Proof-Carrying code architecture

In a Proof-Carrying code architecture (Section 3.5) for a Java-like programming language, the code user (CU) receives the program to be executed from the code producer (CP), in form of .class files. He or she wants to be guaranteed that the program execution is safe with respect to Abstract Non-Interference for a given abstract property of data.

6.3.1 What the code user receives

Our PCC-like architecture follows the approach outlined in [15] (Section 3.5.2). The following components are provided to the code user:

- The program $\mathcal{P}$, as a set of .class files.
- The partition $\{H, L\}$ of data.
- The abstract property $\rho$. It describes the abstraction level of attackers on properties of program classes. Basically, in order to describe $\rho$, it is sufficient to provide an implementation of the abstract dependency calculus (see $\text{ids}(\cdot)$ and $\text{REF}$ in Section 5.2.5). The dependency calculus is the only component of the code certification architecture which is affected by $\rho$.
- An implementation of the ANI($\cdot$) algorithm for bytecode, together with a soundness proof.
- A $\psi$-state $\psi^{\mathcal{P}}(\mathcal{P})$, which is claimed to describe the information flow behavior of $\mathcal{P}$.

6.3.2 What the code user should verify

The code producer provides, together with the program code, the assertion $\text{my program satisfies ANI for } \rho, L \text{ and } H$ (which were made available with the code). This assertion has the form of the $\psi$-state $\psi^{\mathcal{P}}(\mathcal{P})$. The code user needs to check the assertion in order to safely execute the code.
The security policy

The ANI security policy is encoded in $\rho$ and can be fully specified by an algorithm for computing abstract dependencies (essentially, the function $\text{REF}$). We assume that $\rho$ models a security requirement from the user (that is, a property the user considers as necessary in order to accept the program). CU sends $\text{REF}$ to CP, which computes abstract dependencies accordingly. $\text{REF}$ can be considered as a trusted component: CU does not need to check it. The same algorithm is used by CU to verify $\psi^{CP}(P)$.

The boolean function

The code user receives the $\psi$-state $\psi^{CP}(P)$ as a representation of the security properties of $P$. Then, CU should verify:

- that $\Psi_0(P)$, obtained from $\psi^{CP}(P, \text{main})$ with the provided $H$ and $L$, is indeed unsatisfiable (Definition 5.18), that is,

  $$\left( \bigwedge_{x \in L} \neg x \land \psi^{CP}(P, \text{main}) \land \bigvee_{x \in H} x \right)$$

  has no models

- that $\psi^{CP}(P)$ correctly describes the information flow behavior of $P$.

The first task could be accomplished by means of some algorithm for deciding unsatisfiability of formul. However, due to the complexity of such an algorithm, the burden of proving unsatisfiability is left to the code producer, following the Proof-Carrying code style. An UNSAT proof is then attached to $\psi^{CP}(P)$; the code user only needs to verify it.

The validity of $\psi^{CP}(P)$ with respect to $P$ is checked by running, on the user side, the ANI analysis of $P$. The algorithm ANI() for bytecode programs is used; it can be either provided by CP (with a soundness proof) or be possessed by CU (and be a trusted component of the PCC architecture).

Then, the code user needs to execute ANI() on the code. However, he or she does not need to perform the whole (fixpoint) computation: it is sufficient to verify that the provided $\psi$-state $\psi^{CP}(P)$ is indeed the result of executing ANI($P$). To do so, one single iteration of ANI() main loop is needed, that is, an execution of analyze($\text{main}, \psi^{CP}(P)$). If $\psi^{CP}(P)$ happens to be a fixpoint (that is, analyze() does not change the $\psi$-state), then it is accepted as a correct description of $P$. 

6.3 Towards a Proof-Carrying code architecture 111
7

Conclusions

Arguments are to be avoided; they are always vulgar and often convincing.

Oscar Wilde

After all, all he did was string together a lot of old, well-known quotations.

H. L. Mencken (1880 - 1956), on Shakespeare

A static analyzer

The main purpose of this work was to provide a characterization of Abstract Non-Interference which is more likely to be automatizable than the original definition. As pointed out several times throughout previous chapters, the main difficulty is in representing and computing abstract properties.

The ANI analyzer describes the information flow behavior of programs by means of boolean functions, extending to a more realistic programming language previous work which uses the same technique [28]. What makes a difference between this algorithm and an analogous version dealing with standard information flow mainly affects the \( \text{REF} \) function and the calculus of dependencies. If standard Non-Interference is considered, then \( \text{REF}(e) \) comes to be simply the set of identifiers which occur in \( e \). Conversely, abstract dependencies are selective in the choice of relevant identifiers: some of them play no rôle in determining the abstract value of \( e \). Our calculus of abstract dependencies clearly needs to be fully specified and made practical by means of optimization steps and a good choice for the abstract evaluation of expressions. However, no other technique to achieve this result was found in the literature (see the discussion in Section 3.6.2).
In addition, it is shown how abstract dependencies are the first step in defining the promising technique of abstract slicing. Abstract slicing still needs a comprehensive formulation. However, we believe many of the existing frameworks can be reused in order to develop a variant of program slicing for approximated properties of data.

Chapter 4 proposes a notion of Abstract Non-Interference which can be applied to realistic Object-Oriented programs. Aliasing is accounted for through the computation of sharing identifiers. A clever technique is needed not to be overly conservative with respect to aliasing (see Sections 3.6.1 and 5.1.3), that is, to assert the mutual independence of identifiers which cannot share any memory location.

**Code certification**

Some work is still to be done in the direction of a Proof-Carrying code architecture for the certification of Abstract Non-Interference in Java-like programs.

A translation of boolean function analysis has been outlined in Section 6.2. Yet, a complete specification of the PCC architecture must be provided (see Section 6.3.1), in particular, of its components. For example, (i) the representation of the s-state which comes together with the bytecode should be made practical and cheap; (ii) the implementation of \texttt{REF} should be fully specified and optimized; (iii) moreover, the implementation of ANI() should be made part of the architecture, and a suitable representation should be provided for its soundness proof.
A

The abstract dependency calculus: Haskell Implementation

Below, we give the Haskell [62] code for the dependency calculus introduced in Section 5.2. For brevity, we do not include the definitions of the abstract domain (that is, the type ABSVAL and its operators), so that the following code cannot be properly considered as a working implementation.

```haskell
-- a variable is denoted by a numeral index
type VAR = Int

-- atomicity: an abstract value is atomic iff it has no subvalues
-- the type ABSVAL is implicit
atomic :: ABSVAL -> Bool
atomic v = null (subValues v)

-- Environments

-- environments map variables to abstract values
-- updating of an environment at the variable x with v = update :: ENV -> VAR -> ABSVAL -> ENV
update env x v = \ y -> if (y==x) then v else (env y)

-- computes the set of environments env0 which are equal to env on y <> x, and such that env0(x) < env(x)
subs :: ENV -> VAR -> [ENV]
subs env x = map (\ v -> update env x v) (subValues (env x))
```
--- checks if an environment is atomic on each variable ---
--- of an expression --------------------------------------

atomicEnv :: ENV -> EXPR -> Bool
atomicEnv env e = all (\x -> atomic (env x)) (vars e)

--- Abstract evaluation ------------------------------------

--- an expression can be an abstract constant, a var, ---
--- a sum, a product or a projection ------------------------
data EXPR = CONST ABSVAL
    | VARIABLE Int
    | SUM EXPR EXPR
    | PROD EXPR EXPR
    | FST EXPR EXPR
    | SND EXPR EXPR

--- computes the set of variables of an expression ---------
vars :: EXPR -> [VAR]
vars (CONST v) = []
vars (VARIABLE x) = [x]
vars (SUM e1 e2) = union (vars e1) (vars e2)
vars (PROD e1 e2) = union (vars e1) (vars e2)
vars (FST e1 e2) = union (vars e1) (vars e2)
vars (SND e1 e2) = union (vars e1) (vars e2)

--- evaluates an expression in an abstract environment ---
--- (relies on abstract operations on values)--------------
eSharp :: EXPR -> ENV -> ABSVAL
eSharp (CONST v) env = v
eSharp (VARIABLE x) env = env x
eSharp (SUM e1 e2) env =
    aSum (eSharp e1 env) (eSharp e2 env)
eSharp (PROD e1 e2) env =
    aProd (eSharp e1 env) (eSharp e2 env)
eSharp (FST e1 e2) env = eSharp e1 env
eSharp (SND e1 e2) env = eSharp e2 env

--- computes the least upper bound of the evaluation of e
--- (by means of eSharp2) on all the subs of env on x ---
--- if no subinputs exist, top is returned (note that, --
--- usually, bottom is taken as the lub of an empty set;
--- this choice depends on having defined subs by means --
--- of < instead < =, for termination problems)----------
evalSubs :: EXPR -> ENV -> VAR -> ABSVAL
evalSubs e env x = let ch = subs env x in
  if (null ch) then TOP
  else setLub (map (\ env0 -> (eSharp2 e env0)) (subs env x))

-- abstract evaluation of expressions -------------------------
-- unlike eSharp, here the evaluation can take advantage
-- of the evaluation of e on the subinputs of env -----------
eSharp2 :: EXPR -> ENV -> ABSVAL
eSharp2 e env = if (atomicEnv env e) then (eSharp e env)
  else glb (setGlb (map (evalSubs e env) (vars e)))

-- checks if the evaluated expression is atomic ---------------
-- and returns the computed abstract value ------------------
at :: EXPR -> ENV -> (ABSVAL, Bool)
at e env = let ee = (eSharp2 e env) in (ee, atomic ee)

-- computes the union of the sets of relevant variables
-- which are obtained by the subinputs of env -------------
-- if no subinputs exist, returns the whole set of --------
-- variables (which is neutral in the intersection ------
-- performed inside ref3) ---------------------------------
refSubs :: EXPR -> ENV -> VAR -> [VAR] -> [VAR] -> [VAR]
refSubs e env z xs ys =
  let ch = (subs env z) in
    if (null ch) then ys
    else setUnion (map (\ env0 -> snd (ref3 e env0 (union [z] xs) ys)) ch)

-- computes the relevant variables of an expression -------
-- if the expression is constant and atomic (no relevant
-- variables), it is replaced, in the returned value, --
-- by a constant ------------------------------------------
-- ys is the set of vars which are possibly relevant -------
-- (some variables may have been ruled out by previous --
-- computations) -----------------------------------------
ref3 :: EXPR -> ENV -> [VAR] -> [VAR] -> (EXPR, [VAR])
ref3 e env xs ys =
  let (vv, bb) = (at e env) in if bb then (CONST vv, xs)
    else (e, foldr (\ y ->
        \ zs -> refSubs e env y xs zs) ys ys)

-- subexp extraction (only for simplifying the code) -----
sons :: EXPR -> (EXPR,EXPR)
sons (SUM e1 e2) = (e1,e2)
sons (PROD e1 e2) = (e1,e2)
sons (FST e1 e2) = (e1,e2)
sons (SND e1 e2) = (e1,e2)

-- builds a compound expression with e1 and e2, after --
-- the structure of its first parameter (only for --
-- simplifying the code)-------------------------------
recombine :: EXPR -> (EXPR,EXPR) -> EXPR
recombine (SUM ea eb) (e1,e2) = (SUM e1 e2)
recombine (PROD ea eb) (e1,e2) = (PROD e1 e2)
recombine (FST ea eb) (e1,e2) = (FST e1 e2)
recombine (SND ea eb) (e1,e2) = (SND e1 e2)

-- main function-------------------------------------
-- does recursion on the structure of the expression --
-- takes advantage of the substitutions which are --
-- possibly operated by ref3----------------------------
ref2 :: EXPR -> ENV -> [VAR] -> (EXPR,[VAR])
ref2 (CONST v) env ys = ref3 (CONST v) env [] []
ref2 (VARIABLE z) env ys = ref3 (VARIABLE z) env [] [z]
ref2 e env ys =
  let (e1,e2) = sons e in
  let ((ee1,r1),(ee2,r2)) = (ref2 e1 env ys ,
    ref2 e2 env ys) in
  let ee = recombine e (ee1,ee2) in
  ref3 ee env [] (union r1 r2)

-- the most general abstract environment: no information
baseEnv :: ENV
baseEnv = \ i -> TOP

-- calls ref2 on the general environment---------------
ref :: EXPR -> (EXPR,[VAR])
ref e = ref2 e baseEnv []

-- The expression in the example----------------------
exp = (SUM (PROD (CONST POSEVEN) (VARIABLE 1))
    (SUM (PROD (CONST POSEVEN)
    (PROD (VARIABLE 2) (VARIABLE 2)))
    (VARIABLE 3)))
References


34. Roberto Giacobazzi and Elena Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In Patrick Cousot, editor, *Proceed-
122 References


Index

\( \Psi \) function, 101
\# function, 75
\( M_d \) set, 63
\( \Psi_e \) function, 91
\( \Psi_{(bytecode)} \) function, 105
\( \Psi \) function, 83, 91, 100, 106
\( \Psi \parallel \) function, 91
\( T() \) function, 65
\( \Sigma \) (set of traces), 37
\( \psi \parallel \) function, 64
\( \psi \)-state, 85, 88, 108
\( \psi_i \) function, 88
\( \psi \perp \) formula, 89
abstract address, 36
abstract dependency, 4, 36, 82, 108, 111
abstract domain, 15, 19, 37, 43, 45, 56, 68, 113
abstract environment, 69
Abstract Interpretation, i, 3, 11, 15, 26, 44, 73, 87
abstract location, 35
Abstract Non-Interference, i, 3, 4, 26, 38, 39, 43, 44, 55, 59, 63, 82, 97, 106, 108, 111
(non-narrow) ANI, 28
narrow ANI, 28, 40
abstract operation, 4, 43
abstract property, 4, 8, 15, 37, 43, 45, 48, 55, 83, 84, 92, 108, 111
abstract semantics, 64, 83, 101
abstract slicing, 111
abstract value, 19, 27, 43, 56, 105, 111
abstraction function, 17, 37
accept (a program), 87, 106, 108
Access Control, 2, 23, 54
actual parameter, 52, 83, 85
address, 36
administrator, 2
aggregate data, 3
alarm, 36
aliasing, 34, 65, 111
analysis certifier, 32
analysis designer, 57
analysis domain, 57
analytical representation, 44
analyze() procedure, 89, 109
analyser, 73
ANI() procedure, 86, 88, 108, 112
antisymmetry, 11
ascending chain, 12, 21, 88, 89
assembly language, 33
assertion, 35, 108
independence assertion, 35
programmer assertion, 35
region assertion, 35
assignment, 5, 7, 25, 31, 36, 65, 67, 81, 82, 84, 105, 106
(of boolean variable), 82
dangerous assignment, 40, 81
atom, 71
ATOM() predicate, 71
attacker, 2, 3, 8, 25, 26, 51, 55, 82, 108
average, 3
axiomatic semantics, 36
\( B^a \) function, 29
best correct approximation, 20, 68
bijection, 13, 35
binary format, 9, 97
binary relation, 11
block, 48, 99, 107
(boolean variable) assignment, 82
boolean function, i, 40, 63, 80, 82, 89, 98, 106, 111
boolean guard, 38, 83
boolean variable, 39, 80, 89, 98, 106
bottom element, 6, 12
bound
  computational bound, 80
  greatest lower bound, 12, 56
  least upper bound, 12, 56
  lower bound, 12
  upper bound, 12
bounded computation, 87
branch, 38, 47, 83, 99
branching variable, 107
bytecode, see Java bytecode
C language, 33, 34
C++ language, 34, 57
call-by-value, 6
can contain secrets, 81
cardinality, 89
cartesian product, 11
certificate, 23, 31, 39
certification, i, 3, 4, 31, 44, 97, 108, 112
class, 4, 7, 9, 38, 49, 55, 63, 85, 98, 103, 108
  class hierarchy, 56, 80
  class library, 52
  classes as sets, 55
  empty class, 56
  primitive class, 8
  properties as classes, see properties as classes
  singleton class, 8
  subclass, 8, 55
  superclass, 50, 56
types as classes (equivalence), 7
.class file, 9, 97, 107, 108
closure operator, 18
  lower closure operator, see lower closure operator
  upper closure operator, see upper closure operator
code, 1, 4
code consumer, 31
code producer, 31, 39, 108
code user, 4, 9, 31, 39, 44, 97, 107, 108
compiler, 66
compiling, 9, 39, 98, 106
complement, 71
complementation, 19
complete lattice, 12, 19
completeness, 20
complexity, 28, 71, 109
component (of a program), 45
component (of an architecture), 108
composition, 38
compositionality, 84, 99, 106
computable, 43
computational bound, 74, 80
concatenation, 98
concrete domain, 19, 56
concrete property, 16
concrete value, 43
concretization function, 17
concurrency, 7, 50
conditional, 5, 7, 83, 91, 100, 106
conditioned slicing, 28, 95
confidentiality, 1, 4, 23, 26
conservative, 83
consistent, 82
constancy, 35, 84, 105
constant, 10, 46, 47, 68
contains secrets, 81
content (informational), 81
control flow, 48, 67, 83, 99
Control Flow Graph, 48, 99, 107
control flow graph, 29
copyright, 2
correctness, 88, 97
covering, 75
crash, 1
criterion, 92
cycle, 101
dangerous assignment, 40, 81
dangerous flow, 3, 82, 87
data, 1, 3–5, 8, 17, 23, 26, 28, 37, 44, 56, 63, 66, 81, 108
  aggregate data, 3
  forbidden data, 2
  particular data, 3
dataflow equation, 29
debugging, 28
deceptive flow, 27
decidability, 16
declaration, 49, 64
Declassification, 26
decompling, 2
DEF() function (slicing), 29
denotation, 37
dependency, 2, 25, 28, 108, 111, 113
    abstract dependency, see abstract
dependency
concrete dependency, 69
control dependency, 29, 84, 92
data dependency, 4, 29, 66, 92
dependency graph, 67
false dependency, 70
local dependency, 38
descriptor, 9
disjunctive completion, 19
dispatch, 36
domain of parity, 20, 46, 57
domain of sign, 17, 20, 56, 93
downflat domain, 57
download, 4
dynamic, 15
dynamic slicing, 28, 95

edge, 67, 99
Edinburgh Logical Framework, 34
efficiency, 16, 89, 92
environment, 6, 36, 46
equivalence, 98
exception, 7, 38, 48, 50, 101
executable form, 97
execution time, 2
exponential, 71
expression, 5, 7, 65, 67, 83, 93, 106, 111
external user, 97
extraction relation, 35

false alarm, 36
field, 7, 35, 50, 58, 65, 98, 105, 107
    field name, 65
    instance field, 40
    static field, 39, 85
finished, 87
FINISHED predicate, 87
first-order logic, 80

fixpoint, i, 13, 16, 21, 30, 38, 47, 84, 88,
    94, 101, 109
    fixpoint semantics, 16
greatest fixpoint, 13
least fixpoint, 13
fixpoint approximation, 84
flat domain, 58
flow, 3
flow sensitive, 36
Floyd, 33
forbidden data, 2
forbidden flow, 81
foreign functions, 32
formal parameter, 65, 85
formalism, 80
foundational computer science, 16
frame, 10
frame rule, 36
framestack, 10, 102
free member, 2
fresh, 98
fst function, 78
FUN language, 6, 44, 49
function, 6, 12, 37, 71
    arity, 12, 91
    bijection, 35
    bijective, 13
    composition, 13, 17
    continuous, 12, 21
    extensive, 12, 17, 87
    idempotent, 12
    image, 12, 21
    injective, 13
    monotone, 12, 18, 74, 88
    partial, 12
    pointwise ordering, 13
    reductive, 12, 17
    surjective, 13
    total, 12
functional language, 5, 15
Galois connection, 17, 26, 65
garbage collection, 49
graph, 67, 88, 99
ground, 105
guard, 38, 83
H modifier, 54
halting problem, 16
Haskell, 33
Haskell language, i, 75, 113
heap, 10, 35, 39, 85
high-level (language), 9, 33, 39, 106
high-level (program structure), 100
higher-order, 15, 46
Hoare logic, 35, 40, 66
   Hoare triple, 35, 40
I/O, 49
identifier, 7, 29, 34, 55, 64, 81, 82, 89,
   90, 105, 111
   dotted identifier, 7
   sharing identifier, see sharing identifier
identity domain, 28
ids() procedure, 108
IMP language, 5, 25, 29, 49, 68, 82, 84
imperative language, 5, 36, 49, 82
implicit flow, 83, 90
incompleteness, 73
increasing chain, 12
independent identifiers, 34
indistinguishability, 38
indistinguishable, 46
Information Flow, i, 3, 4, 8, 23, 30, 34,
   38, 44, 49, 63, 80, 82, 92, 97, 98,
   105, 108, 111
   direct flow, 25
   implicit flow, 83, 90
   indirect flow, 25
information system, 1, 2
informational content, 81
inheritance, 50, 102
   multiple inheritance, see multiple inheritance
injection, 13
input, 2, 5, 25, 27, 28, 37, 44, 49, 81, 84,
   90, 98
instance, 10
instruction, 48, 98
instruction set, 9
interface, 50
interpreter, 97
interprocedural, 36
invariant, 33, 41
isomorphism, 13, 19
iteration, 83, 87, 88
Java bytecode, i, 4, 9, 26, 39, 44, 97,
   106, 108, 112
   instructions
   aconst_null, 102
   aload, 99, 103
   areturn, 105
   astore, 99, 103, 105, 106
   dup, 99, 102
   getfield, 99, 103
   getstatic, 103
   goto, 99, 100, 107
   if_acmpeq, 99, 100
   if_acmpneq, 99
   ifeq, 99
   if_icmpeq, 99
   ifnonnull, 99, 100
   invokevirtual, 99
   ldc, 105
   new, 99, 105
   nop, 102
   pop, 102
   putfield, 99, 103, 107
   putstatic, 103
   return, 105
   swap, 102
   non-jump instruction, 99
Java language, i, 4, 7, 9, 26, 34, 50, 56,
   80, 97, 108
Java compiler, 97
Java Virtual Machine, 9, 38, 97, 107
JL language, 7, 63, 97
judgement, 35, 47
jump, 99
L modifier, 54
label, 99
lambda-abstraction, 6
lambda-calculus, 6, 26, 44
lambda-expression, 45
lattice, i, 12
lifting, 20, 48
list, 33
local dependency, 38
local variable, 98
location, 34, 50, 65, 82, 111
   abstract location, see abstract location
LOCS function, 65
Index 129

logic language, 15
logical equivalence, 81, 98
loop, 5, 7, 34, 83, 93
low-level (language), 4, 39
lower closure operator, 18

machine register, 33
machine-dependent, 97
main() procedure, 52, 85, 109
main loop, 88, 109
map, 12, 17
member, 9, 63
  static member, 85
memory, 2, 6, 9, 34, 39, 65, 111
    allocation/deallocation, 49
    memory configuration, 37
method, 7, 9, 36, 38, 50, 58, 63, 85, 98
  instance method, 50, 63
  method body, 64, 99
  method definition, 63
  method instance, 63, 83, 91, 100
  method invocation, 7, 64, 83, 87, 92
  method name, 63
  method overriding, 39, 57, 63
  method signature, 63
  method summary, 36
  static method, 39, 52, 65, 85
method stack, 85
mistrustful, 97
ML language, 26, 33
model, 82, 89, 109
modifier, 54
  private modifier, see private modifier
  protected modifier, see protected modifier
  public modifier, see public modifier
monomorphic, 6, 47
Moore closure, 12, 19
Moore family, 12, 19
multiple inheritance, 56
mutual recursion, 85

native code, ii, 97
Non-Interference, 3, 8, 24, 30, 38, 44, 49, 55, 82
  (standard) NI, 26, 111
non-jump, 48

object, 36, 39, 50, 55, 66, 98
  object creation, 7, 36, 39, 49, 57, 105
Object Oriented, 111
Object-Oriented, i, 4, 7, 34, 49, 55, 65, 82
observable behavior, 3
observable dependency, 37
observational power, 3, 26, 45, 56, 82
observer, 2, 50, 84
occurrence, 68
one-state semantics, 35
opcode, 10
open source, 2
operand stack, 10, 38, 48, 98
  top element, 98
operator, 98
optimization, 32
order isomorphism, 13
ordering, 16, 74, 88
output, 2, 5, 27, 44, 49, 81, 84, 98
overriding, 39, 57, 63

pair, 44
parameter
  actual parameter, see actual parameter
  formal parameter, see formal parameter
parameter, 10, 39, 64, 89, 103
parity, 46, 57
partial order, 12
partially ordered set, 12
particular data, 3
partition, 71, 108
partition (of domain), 41
partitioning domain, 71
passive, 2
password checking, 26
path, 38, 67, 83
paying member, 2
pointer aliasing, see aliasing
pointwise, 74
pointwise ordering, 13
polymorphism, 50
polyvariance, 36
pop, 10
postcondition, 33, 36
powerset, 11, 56, 89
precision, 8, 17, 27, 41, 44, 46, 55, 70, 93
precondition, 33, 36
predicate, 35, 56, 80, 87
prevalue, 48
primitive domain, 48
private, 2, 25, 26, 30, 45, 49, 81
private modifier, 54
privilege, 2
privileges, 23
procedural, 48
procedure, 7, 49, 71
program data, see data
program point, 29, 37, 64, 66, 81, 84, 92
program slicing, see slicing
program state, 6
proof system, 40
proof validator, 31
Proof-Carrying code, i, 4, 31, 97, 108, 112
PCC architecture, 108
propagation, 24, 63, 66, 80, 84
properties as classes, 4, 9, 55, 80
property, 8, 27, 35, 63, 108
precision, see precision
program property, 17
properties as classes, see properties
as classes
properties as sets of values, 17
property transformer, 20
security property, see security property
undecidable property, 16
protected, 3, 23
protected modifier, 54
prover, 28, 35
public, 2, 25, 26, 30, 44, 49, 55, 81
public modifier, 54
push, 10, 85
$R^{S'}_S$ (i) function, 29
reached (fixpoint), 88
recursion, 6, 44, 77
mutual recursion, see mutual recursion
recursive call, 87, 91
reduced product, 19
Ref function, 80, 84, 93, 100, 105, 106, 108, 111, 112
Ref() function (slicing), 29, 92
Ref2 function, 76
Ref3 function, 76
reference, 11, 98, 103
reflexivity, 11
reject (a program), 39, 87
relation, 11, 88
Relaxed Non-Interference, 26, 44
relevant, 39, 69, 82, 84, 92, 105, 111
renaming, 40
requirement, 3
restriction, 75
result value, 105
Ref variable, 81, 83, 105, 106
return value, 7
reverse engineering, 28
runtime, 49, 58, 64, 72, 83, 84
Runtime Constant Pool, 98
runtime system, 32
safe, 31, 34
safe program, 107
safe software, 97
safety, 97, 108
safety proof, 31
salary, 3
same role, 106
satisfiability, 82, 87, 109
secret, 63, 80, 103
security, 23
high-security, 25, 26
Language-based security, 23
low-security, 25, 26
security checking, 97
security policy, 1, 4, 23, 31, 58, 84, 97, 106, 108
security property, 1, 3, 4, 44, 87, 97, 108
security requirement, 1, 3, 4, 44, 87, 97, 108
security signature, 39
security type, 44
semantic approximation, 15
semantic slicing, 36
semantics, 2–4, 6, 25, 26, 50, 55
abstract semantics, 16, 22, 64, 83, 101
axiomatic semantics, 36
Index

collecting semantics, 22

denotational semantics, 6, 43

fixpoint semantics, 16

input-output semantics, 22, 50

one-state semantics, 35

trace semantics, 22, 37, 50

two-state semantics, 35

semantics: abstract semantics, 68

separation, 3, 66

Separation Logic, 36

sequential, 48

sequential combination (of boolean functions), 98

sequential composition, 84

set (representation), 45

si function, 66, 85, 103, 106

sharing, 34, 66, 82, 84

sharing identifier, 66, 84, 111

side effect, 80, 83

sign, 57, 68, 93

signature, 63

skip, 85

slice, 92

slicing, 28, 38, 67, 92, 111

abstract slicing, see abstract slicing conditioned slicing, 28, 95
dynamic slicing, 28, 95
slice, 28
slicing criterion, 28
static slicing, 28


source code, i, 2, 4, 9, 82, 97, 106

stack, 10, 48, 80, 85, 98

framestack, 102

standard deviation, 3

state, 10, 25, 35, 37, 102

statement, 5, 7, 25, 28, 35, 64, 67, 81, 83, 91, 92, 98, 106

statement schema, 91

static, 10, 15, 101, 103

Static Analysis, 1, 4, 16

static analyzer, 32, 63

static field, 85

static member, 85

static method, 85

static slicing, 28

store, 35

strongest postcondition, 36

subclass, 36, 105

subprogram, 49, 63

subroutine, 80

∥ function, 75

subscript, 81, 84, 98

top element (of a stack), 98
top element, 7, 12, 44

trace semantics, 37

transitive flow, 25

transitivity, 11

triple, 35

trusted, 108

trusted component, 31

truth value, 82

Turing-equivalence, 16, 43
two-state semantics, 35
type, 5, 15, 34, 44, 49, 64, 65, 93

primitive type, 8, 99

security type, 26, 44
type analysis, 44
type cast, 39
type checking, 6, 55, 57
type constructor, 6
type environment, 64, 83, 105
type inference, 46
type safety, 49
type system, 6, 15, 22, 25, 38, 47
types as classes (equivalence), 7
Index

typing rule, 47
typing rules, 34
union type, 33

undecidability, 4, 48
universal language, 43
universal quantifier, 43, 48
UNSAT (proof), 109
untrusted, 2, 4, 31
untrusted user, 3, 55
upper closure operator, 18, 27, 35, 45, 56, 68

validation, 44
value, 6
variable, 5–7, 17, 25, 27, 36, 45, 49, 64, 66, 81, 82, 98
global variable, 39
local variable, 7, 10, 38, 98

private variable, 25
public variable, 25
variable declaration, 49, 64
variable renaming, 40
variable substitution, 81, 90
VARS() function, 29, 84, 107
VC predicate, 32
VC generator, 33
verification condition, 32
vertex, 67, 99
visitor, 2

Web, 2, 4
web, 23, 31
widening, 21, 84
WIDENING operator, 89
World Wide Web, 4
wrong value, 6
Sommario

Questo lavoro è organizzato in sette capitoli. L’introduzione, Capitolo 1, illustra il contesto in cui la ricerca si colloca, ed evidenzia l’importanza delle proprietà di sicurezza nella pratica attuale dello sviluppo ed analisi del software.

Il Capitolo 2 introduce i diversi linguaggi di programmazione utilizzati nei capitoli successivi (tranne Haskell, usato per implementare il calcolo delle dipendenze incluso nell’Appendice A). Questo capitolo contiene, inoltre, i riferimenti necessari all’teoria dei reticoli e dei punti fissi.

La definizione originale di Non-Interferenza Astratta e le teorie su cui essa si basa (in particolare, l’Interpretazione Astratta e i fondamenti dell’analisi dei flussi di informazione) sono presentati nel Capitolo 3. Poiché il nostro interesse è rivolto alla certificazione del codice, introduciamo la promettente architettura Proof-Carrying code (codice dotato di una prova di correttezza).

Il Proof-Carrying code è l’obiettivo dell’architettura delineata nel Capitolo 6. Anche il Capitolo 3 presenta, nella sua ultima sezione, ricerche recenti che sono importanti per questo scopo. Un’attenzione particolare è rivolta all’analisi di sicurezza per i linguaggi orientati agli oggetti. Ci occuperemo inoltre dell’attuale ricerca sulla modellazione di proprietà astratte.

La parte principale di questo lavoro inizia al Capitolo 4, dove si discute esaurientemente il problema di rappresentare proprietà di sicurezza relative al flusso di informazioni. Nella prima sezione viene presentato un approccio basato su tipi di sicurezza, che costruisce un passo iniziale verso l’automatizzazione della Non-Interferenza Astratta. Successivamente, si discute un approccio realistico ad ANI. Il capitolo contiene una definizione completa di ANI per un linguaggio simile a Java; questo approccio sfrutta i meccanismi propri del linguaggio per controllare le proprietà di sicurezza.

Questa definizione è applicata nel Capitolo 5, che sviluppa un’analisi di ANI su codice sorgente, ottenuta attraverso l’uso di funzioni booleane. L’analisi del flusso di informazione si basa su un calcolo delle dipendenze astratte sui dati; esse sono calcolate da un algoritmo descritto in questo capitolo. Le dipendenze sono inoltre considerate in relazione al Program Slicing.
L’analisi del codice sorgente viene tradotta, nel Capitolo 6, al linguaggio Java bytecode, in vista della certificazione di programmi all’interno di un’architettura PCC.

Il Capitolo 7 contiene le conclusioni e direzioni future di ricerca.